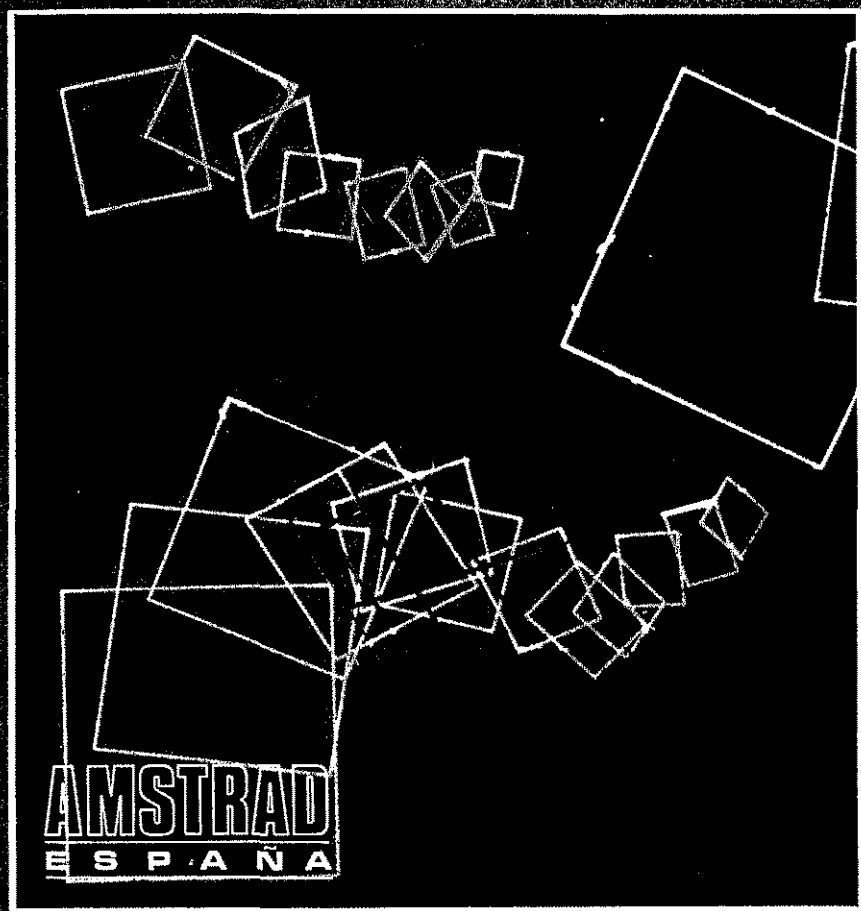


# **CODIGO MAQUINA PARA PRINCIPIANTES CON AMSTRAD**



**Steve Kramer**



# CÓDIGO MÁQUINA PARA PRINCIPIANTES CON AMSTRAD

Steve Kramer



# CÓDIGO MÁQUINA PARA PRINCIPIANTES CON AMSTRAD

edición española de la obra

## MACHINE CODE FOR BEGINNERS ON THE AMSTRAD

Steve Kramer

publicada en castellano bajo licencia de

MICRO PRESS

Castle House

27 London Road

Tunbridge Wells, Kent

### Traducción

Pablo de la Fuente Redondo

Director del Centro de Proceso de Datos

Universidad de Valladolid

### Revisada por

Jesús Rojo García

Profesor de Matemática Aplicada

Escuela T. S. de Ing. Industriales de Valladolid

INDESCOMP, S.A.

Avda. del Mediterráneo, 9

28007 Madrid

© 1984 Steve Kramer

© 1985 Indescomp, S.A.

Reservados todos los derechos. Prohibida la reproducción total o parcial de la obra, por cualquier medio, sin el permiso escrito de los editores.

ISBN 84 86176 24 7

Depósito legal: M-10847-85

impresión:

Gráficas EMA. Miguel Yuste, 27. Madrid

Producción de la edición española:

Vector Ediciones. Gutierrez de Cetina, 61, Madrid. (91) 408 52 17

# Contenido

1. Introducción .....	1
2. Qué es y para qué sirve el código de máquina .....	3
3. Primeras nociones .....	7
4. Diagramas de flujo .....	13
5. Primeras instrucciones en código de máquina .....	17
LD, CALL, RET, JP, JR	
6. Aritmética elemental! .....	39
ADD, ADC, SUB, SBC, DEC, INC	
7. Indicadores, condiciones y decisiones condicionadas .....	59
CP, Z, NZ, C, NC, M, P, PE, PO, CCF, SCF, DJNZ	
8. Operaciones lógicas .....	71
AND, OR, XOR, CPL, NEG	
9. Utilización de la pila .....	79
PUSH, POP e instrucciones con el SP	
10. Instrucciones que trabajan con un solo bit .....	87
SET, RES, BIT	
11. Rotaciones y desplazamientos .....	95
RL, RLA, RLC, RLCA, RLD, SLA, SLL, RR,	
RRA, RRC, RRCA, RRD, SRA, SRL	
12. Búsquedas y transferencias automáticas .....	113
LDD, LDDR, CPD, CPDR, LDI, LDIR, CPI, CPIR	
13. Comunicación con el exterior .....	125
IN, OUT	
14. Otras instrucciones .....	129
15. Consejos sobre cómo utilizar el sistema operativo .....	141
Apéndices	
A Conjunto de instrucciones del Z80 .....	145
B Cargador HEX .....	155
C Conversión de HEX a DECIMAL para el byte más	
significativo .....	157
D Conversión de HEX a DECIMAL para el byte menos	
significativo .....	159
E Conversión de HEX en complemento a 2 a DECIMAL .....	161
F Mapa de pantalla del Amstrad .....	163
G Dirección de las rutinas más usuales del sistema operativo. ...	167



## Introducción

El Amstrad CPC464 es probablemente la novedad más interesante en materia de ordenadores domésticos tras la aparición del Spectrum. Su BASIC está dotado de funciones avanzadas que hasta ahora sólo se incluían en máquinas de precio muy superior; además, en cuanto a posibilidades de ampliación a precio razonable, no tiene nada que envidiar a los demás ordenadores de su categoría.

Ahora bien, la diferencia fundamental entre éste y otros ordenadores, por lo que al programador concierne, está en la decisión de Amstrad de publicar su exhaustiva documentación sobre el sistema operativo. Este hecho, sin precedentes en la industria de los ordenadores domésticos, ofrece la posibilidad de aprender programación en código de máquina por la vía fácil y de obtener resultados casi inmediatos utilizando rutinas del sistema operativo.

Superado queda el círculo vicioso en que antes nos encontrábamos: si no entiendo el código de máquina, no puedo utilizarlo, y por lo tanto nunca podré averiguar cómo funciona en mi ordenador, pues no sé cómo hacer que éste responda.

Este libro se dirige a los principiantes que deseen aprender a programar en código de máquina en el Amstrad CPC464. Empezaremos por examinar los conceptos básicos de programación en código de máquina, explicando las instrucciones reconocibles por el microprocesador Z80 y cómo utilizarlas. A lo largo del libro describiremos también algunas rutinas del sistema operativo.

Dos personas totalmente noveles en código de máquina me han servido de banco de pruebas en la elaboración de este libro; sus preguntas y observaciones forman la base de la estructura de la obra. Su ayuda ha sido especialmente valiosa para asegurar que no se omitiera ninguna información o explicación que, aunque obvia para el experto, para el principiante pudiera ser clarificadora. Estas omisiones suelen ser las que dejan desconcertado al principiante; algo así como decirle a un forastero que la calle Desengaño está junto a la Gran Vía. ¿De qué le sirve esa información si no sabe dónde está la Gran Vía?

Daremos algunos pequeños programas en BASIC con los que se podrá in-

introducir programas en código de máquina, así como examinar y modificar el contenido de zonas de la memoria. No obstante, sugerimos al lector que haga lo posible por adquirir el programa ensamblador/desensamblador de Amsoft. Esto le permitirá introducir los programas empleando los códigos nemotécnicos (una especie de abreviaturas de las instrucciones que entiende el Z80) en lugar de números; además, con un ensamblador, las modificaciones de los programas son más sencillas y las instrucciones en sí son más próximas a BASIC.

Evidentemente, es posible leer este libro de principio a fin de una sentada. Pero no lo recomendamos. El código de máquina es un tema potencialmente tan confuso, y son tantos los conceptos que se manejan, que lo conveniente es que el lector se siente ante su ordenador e introduzca y ejecute los programas que van apareciendo en cada capítulo, y que no pase al capítulo siguiente mientras no esté seguro de haber comprendido su funcionamiento.

Hemos utilizado ampliamente el sistema operativo de la máquina, lo que hace posible ver inmediatamente los resultados de los programas. Las rutinas del sistema operativo están excelentemente documentadas en la publicación "Amstrad Firmware Specification (Soft 158)". Aunque este texto será totalmente ininteligible para el lector en este momento, no debería dudar en incorporarlo a su biblioteca en cuanto haya terminado de leer este libro.

El microprocesador Z80 es uno de los más utilizados en los ordenadores domésticos y, hasta hace poco tiempo, también en los ordenadores profesionales. Para él se ha escrito la más amplia variedad de programas existente en el mercado, utilizable a través del sistema operativo CP/M, el cual está disponible en disco para los ordenadores Amstrad. Además, el Z80 está siendo incluido como segundo microprocesador en ordenadores profesionales, y como opción en el BBC, el Commodore 64, el Apple y otros. Así pues, los conocimientos que el lector va a adquirir en este libro le servirán también para programar ordenadores de muchas otras marcas.



## 2

### Qué es y para qué sirve el código de máquina

El microprocesador del Amstrad es una criatura básicamente ignorante. Desde luego, ejecuta muy bien todos los programas de BASIC y hace su trabajo a la perfección, pero ello no significa que el Z80 sea inteligente. Lo que hace que la máquina parezca tan hábil es el *firmware*, esto es, los programas que están grabados permanentemente en la memoria del ordenador y entran en funcionamiento en cuanto se enciende la máquina. En el Amstrad no ampliado estos programas son un sistema operativo y el intérprete de BASIC.

El sistema operativo se ocupa de tareas tales como examinar el teclado para averiguar si se ha pulsado una tecla, leer datos de la cinta o escribir un carácter en la pantalla. El lector puede imaginarlo como organizador de todas las comunicaciones, sin el cual no sería posible saber si el ordenador está encendido o apagado ya que no se le podría suministrar información ni él podría reaccionar ante ningún estímulo.

El intérprete de BASIC hace justamente lo que su nombre sugiere: convertir BASIC en un lenguaje comprensible para el Z80. Imagine el lector que le decimos que abra el libro por la página 35. Fácil, ¿verdad? Pero ¿qué ocurre si le decimos que 發.? Empiezan los problemas; no sólo no sabrá qué tiene que hacer, sino que incluso puede no reconocer la forma de la instrucción.

Esto es más o menos lo que le ocurriría al Z80 si le pidiéramos que ejecutase una instrucción de BASIC. El microprocesador no entiende BASIC; pero no es sólo eso. La palabra china que hemos citado utiliza sólo un símbolo, pero para transcribirla a nuestros caracteres son necesarios varios: "tsung". La transcripción tampoco nos ha servido de mucho; "tsung" significa: sembrar semillas sin antes arar la tierra. El microprocesador experimenta las mismas dificultades si le damos una orden en BASIC; una instrucción de BASIC representa muchas veces gran número de instrucciones en código de máquina y, lo que es peor, los caracteres utilizados por BASIC no pueden ser entendidos por el microprocesador, que solamente reconoce dos estados: 1 y 0 (*on/off*, encendido/apagado, etc.).

Afortunadamente, los ceros y los unos se agrupan de ocho en ocho, lo que da 256 combinaciones diferentes posibles. Son estas combinaciones las que

utilizamos como códigos de máquina. Podríamos considerarlas como análogas al carácter chino que vimos antes.

Pero no termina aquí la lista de nuestros problemas. Puesto que un carácter representa una palabra completa y sólo hay 256 combinaciones posibles, podría parecer que el vocabulario de! Z80 está limitado a tan sólo 256 palabras. Esto es básicamente correcto pero, al igual que en los lenguajes ordinarios, hay palabras compuestas.

Por un lado, hay palabras cuyo significado cambia cuando se presentan asociadas: no es lo mismo "tio vivo" que "tio vivo" por ejemplo. Además el sentido de una palabra puede cambiar radicalmente mediante el empleo de prefijos: es el caso de "justicia" e "injusticia", o de "venido", "avenido", "desavenido" y "revenido", o de muchos otros ejemplos. Estas técnicas se emplean también para proporcionar al microprocesador mayor variedad de palabras. A pesar de todo, el vocabulario es muy limitado.

La limitación no afecta a la cantidad de conceptos que puede reflejar el vocabulario, sino a la cantidad de palabras que se necesitan para expresarlos.

Generalmente se suelen necesitar varias instrucciones en código de máquina para realizar lo mismo que con una instrucción de BASIC. En cambio, prácticamente no hay limitaciones en cuanto a la forma en que deben ir ordenadas las instrucciones en código de máquina. Es más, en algunos casos el código de máquina puede requerir menos instrucciones que BASIC para una misma tarea.

El intérprete BASIC debe comprobar la validez de cada una de las instrucciones, traducirlas a instrucciones de código de máquina para que el microprocesador pueda ejecutarlas, comprobar ciertos resultados y archivarlos para utilizaciones posteriores. Todas estas cosas llevan mucho tiempo. Por el contrario, con el código de máquina no deben verificarse los posibles errores, no hay que traducir las instrucciones y no se crea un almacén de datos salvo que se le pida expresamente al microprocesador.

Para comprobar el ahorro de tiempo, teclee el siguiente programa en BASIC. (Antes de hacerlo apague el ordenador y vuelva a encenderlo para asegurarse de que está, por así decirlo, "virgen".) Observe que empleamos el símbolo ? en lugar de PRINT para ganar tiempo.

```
10 MM = 43903
20 MEMORY 43799
30 FOR N = 43800 TO 43809 : READ D : POKE N,D : A
  = A + D : NEXT
40 IF A <> 1338 THEN CLS : PEN 3 : PRINT "ERROR EN
  DATA" : PEN 1 : EDIT 90
```

```

50 INPUT "PULSE ENTER PARA EMPEZAR";A : B = 255
60 PRINT "A";: B = B - 1 : IF B <> 0 THEN 60
70 PRINT
80 CALL 43800
90 DATA 6,255,62,65,205,90,187,16,251,201
100 END

```

Cuando haya introducido el programa, ejecútelo con el comando RUN. Si lo que aparece en pantalla es la línea 90 en modo de edición, lo que ocurre es que se ha equivocado al teclear los datos de esta línea; corrija entonces la línea vuelva a ejecutar el programa. Si ya no hay errores, aparecerá en pantalla el mensaje 'PULSE ENTER PARA EMPEZAR'.

A! pulsar dicha tecla, la línea 60 hará que se escriba 255 veces la letra 'A'; a continuación, línea 80 llama a la rutina en código de máquina que el programa ha cargado con la sentencia POKE de la línea 30; esta rutina tiene por efecto escribir otras 255 veces la letra 'A'. Compare la velocidad de estas dos maneras de hacer lo mismo. El programa no tiene nada de apasionante, pero le demostrará la rapidez del código de máquina.

La rutina en código de máquina ha ocupado 10 caracteres (que son los que figuran en la línea del DATA), el último de los cuales, el 201, sirve para ordenar a la rutina que retorne a BASIC. El programa equivalente en BASIC ha ocupado 37 caracteres, sin contar el número de línea; incluso sin blancos innecesarios no ocuparía menos del equivalente a 25 caracteres de código de máquina.

Para comprobar la longitud que ocupa realmente la línea 60, añada al programa las líneas

```

110 B=0:FOR N=520 TO 639:A=PEEK(N)
120 IF B=0 THEN PEN 2:PRINT:PRINT N;
130 PEN 3:PRINT USING "####";A;
140 IF A>32 AND A<129 THEN PEN 1:PRINT
CHR$(A);:GOTO 160
150 PRINT " ";
160 B=B+1:IF B=5 THEN B=0
170 NEXT: PEN 1:END

```

y ejecútelas con el comando RUN 110. La pantalla mostrará en color rojo los valores que ocupan las posiciones de memoria entre la 520 y la 639; cuando el valor representa un carácter, éste aparece en amarillo a su derecha. Los números en azul corresponden a la primera dirección de memoria de la línea. La línea 50 se reconoce por el mensaje "PULSE ENTER PARA EMPEZAR". A continuación viene la línea 60. El número de línea está donde aparece 0 60 en rojo seguido de < en amarillo y de 0 en rojo; el número de línea es el 600 y el número que aparece antes del primer 0 es el número de caracteres de la línea.

Podemos observar que sólo las cadenas literales, como "A", se almacenan como las escribimos. Los demás caracteres son codificados por el intérprete a una forma que le permita un manejo más fácil. Cada vez que se ejecuta el comando LIST, el intérprete debe decodificar el texto para dejarlo en la forma en que lo hemos escrito.

La conclusión que se obtiene de todo esto es que un programa en código de máquina no sólo es más rápido, sino también más económico de almacenar. Estas son las dos ventajas principales de programar en código de máquina. De hecho, un programa en BASIC puede ser unas cien veces más lento que su equivalente en código de máquina.

Por el contrario, las desventajas consisten en que los programas son prácticamente incomprensibles y, por tanto, difíciles de depurar, y suelen requerir mayor número de instrucciones que sus equivalentes en BASIC o en otro lenguaje de alto nivel.

Se puede mejorar la comprensión de los programas en código de máquina utilizando ensambladores y desensambladores, de los que hablaremos en el próximo capítulo. El problema de la cantidad de instrucciones no es normalmente resoluble, pero el Amstrad CPC 464 tiene la ventaja de que permite utilizar las rutinas de su sistema operativo. La información que Amstrad proporciona sobre estas rutinas le permitirá utilizarlas rápidamente, de manera que en realidad buena parte de sus programas ya ha sido escrita de hecho por Locomotive Software al desarrollar el sistema operativo del ordenador.

### 3

## Primeras nociones

Antes de introducirse en el código de máquina, es necesario conocer algunos conceptos, aunque sea de manera elemental; comenzaremos por explicar brevemente estas nociones.

### Hexadecimal y binario

Son dos sistemas de numeración: el binario en base 2, y el hexadecimal en base 16. El lector posiblemente conocerá ya el sistema binario y no le parecerá muy práctico para realizar operaciones. Sin embargo, es el único método que puede utilizar el ordenador. Como el microprocesador sólo reconoce dos estados, encendido y apagado (correspondiendo 1 a encendido y 0 a apagado), debe trabajar en sistema binario.

Cada cifra binaria, o *bit* para abreviar (de *binary digit*), posee un valor relativo que depende de su posición. Ocurre como con el sistema decimal, donde hay la cifra de las unidades, la de las decenas, la de las centenas, etc. En el sistema binario cada cifra puede tener sólo el valor uno o cero, luego los valores relativos a la posición deben ser reducidos. Si utilizásemos los mismos valores que en el sistema decimal sólo podríamos representar los números cero, uno, diez, once, cien, ciento uno, etc.

El Amstrad almacena la información en conjuntos de 8 bits; cada uno de ellos es un *byte* (se pronuncia 'bait'). También maneja grupos de dos bytes o 16 bits: las denominadas *palabras*. En una palabra, los valores relativos correspondientes a los diferentes bits son los siguientes:

BIT NUMERO															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
32768	16384	8192	4096	2048	1024	512	256	128	64	32	16	8	4	2	1

Con este sistema de numeración, una palabra puede representar con los símbolos 0 y 1 cualquier número comprendido entre 0 y 65535. Observe que el bit menos significativo se numera como bit 0.

Muchas veces hay que representar números negativos. Vamos a ver lo que ocurre cuando restamos a 0 el número 1 para obtener -1. Por brevedad lo haremos sólo con un grupo de 4 bits; se tiene que

```

0000

-1  0 - 1 = 1 y llevamos 1

-1  0 - 1 = 1 y llevamos 1

-1  0 - 1 = 1 y llevamos 1

-1  0 - 1 = 1 y llevamos 1

1111

```

luego la respuesta es el número binario 1111, que es el decimal 15. Utilizando 8 bits o 16 bits hubiésemos obtenido 255 o 65535, respectivamente.

Cuando el resultado de una resta es un número negativo, ocurre siempre que el bit más significativo (el de la izquierda) se coloca a 1. Estos nos da la pista de cómo se representan los números negativos.

Cuando se usan números negativos, se utiliza el convenio de que el bit más significativo representa el signo: 1 para el signo menos y 0 para el signo más. Esto cambia el intervalo de los números que podemos representar. Con 16 bits los números van de -32768 a +32767; con 8 bits, de -128 a +127. Para cambiar un número de signo el procedimiento consiste en cambiar los unos por ceros, y viceversa, y finalmente sumar 1. Esta técnica de representación es la que se denomina "de complemento a dos".

En nuestros programas deberemos emplear, dependiendo del caso, la representación binaria normal sin signo o la representación en complemento a dos. Mencionaremos en cada instrucción el tipo de representación requerido.

El ensamblador GENS permite utilizar números binarios; éstos debe ir precedidos del símbolo %.

Pero, ¿por qué el sistema hexadecimal? Para el ordenador no representa ningún problema trabajar con ceros y unos, pero para nosotros constituye una enorme dificultad. Normalmente el sistema decimal será el que utilizaremos con menor dificultad, pero en ciertas ocasiones nos será más fácil razonar en binario. Por ejemplo, para cargar un byte de manera que cada medio byte represente el número decimal 9, es más fácil trabajar en binario. Como

$1*8+0*4+0*2+1*1=9$ , 9 equivale a 1001, luego lo que necesitaremos tener es 1001 1001; el valor decimal es entonces

$$1*128 + 0*64 + 0*32 + 1*16 + 1*8 + 0*4 + 0*2 + 1*1$$

o sea, 153. Sorprendido, ¿verdad?

En medio byte se pueden almacenar números entre el 0 y el 15, es decir, un total de 16 números. Para trabajar con números binarios es cómodo agruparlos por medios bytes, utilizando así el sistema de numeración en base 16 o hexadecimal. En el ejemplo anterior hubiésemos dicho que había que cargar el número hexadecimal 99, así de sencillo.

Este sistema necesita 16 cifras diferentes. Las primeras son las que van del 0 al 9; para las restantes no se emplean nuevos símbolos, sino que se utilizan las primeras letras del alfabeto. La letra A representa en número decimal 10, la B el 11, y así sucesivamente hasta la F, que representa el 15.

Otro problema que hay que resolver es el de señalar de alguna manera que un número está en hexadecimal, para que no se confunda con uno decimal.

Lamentablemente, no existe para ello ningún convenio que se emplee con generalidad. El Amstrad utiliza el símbolo &, el Firmware Specification Manual utiliza £ y el ensamblador GENS utiliza #, otros ensambladores utilizan una h minúscula o mayúscula.

En este libro los números hexadecimales irán seguidos de la letra minúscula h, excepto en los listados del ensamblador GENS, en los que aparecerán precedidos de #.

## ASCII

ASCII es la abreviatura de American Standard Code for Information Interchange, que es un código (el más utilizado) para representar caracteres alfabéticos, numéricos y de control mediante números. Este código está impreso en el apéndice III de la Guía del Usuario de Amstrad.

## Dirección

Es un número que se utiliza para referenciar las posiciones de memoria. Cada posición de memoria posee una dirección; se comienza por la 0 para la primera posición y se llega hasta la 65535 (FFFFh). Las direcciones se suelen dar en hexadecimal. Casi todos los ensambladores dan en la primera columna de sus listados la dirección en la que se coloca cada instrucción.

## Ensamblador

Hemos hablado varias veces de ensambladores, pero ¿qué es un ensamblador? Vamos a explicarlo.

Un *ensamblador* es un programa que nos permite crear programas en código de máquina escribiendo las instrucciones en forma descriptiva y fácil de recordar en lugar de hacerlo con ceros y unos. Los códigos que sirven para representar así las instrucciones se llaman *códigos nemotécnicos*. El ensamblador nos permite escribir los programas en esta forma y, cuando hemos terminado, los traduce (los *ensambla*) a ceros y unos, que es lo que entiende el microprocesador.

Normalmente los ensambladores disponen también de un *editor*, que permite realizar con facilidad la escritura y corrección del texto de los programas. Si *no* fuera por esta ayuda habría que reescribir completamente el programa cada vez que se encontrase un error en alguna de las instrucciones.

El programa que se escribe con el editor se denomina *programa fuente*; es un programa que no se puede ejecutar mientras no se lo haya ensamblado con éxito. El programa fuente se puede guardar en cinta para su utilización posterior. El programa ya ensamblado, que es el ejecutable, se denomina *programa objeto o código objeto*.

El programa objeto también puede ser grabado en cinta, bien sea con el comando T del ensamblador GENS o desde BASIC. Para grabar en cinta desde BASIC un programa objeto, se utiliza el comando SAVE, cuyo formato es

SAVE "nombre",B,dirección inicial,longitud,punto de entrada

El punto de entrada es la dirección de memoria en la que comenzará la ejecución del programa cuando éste sea cargado con el comando "RUN". Si no se ha especificado esta dirección y este utiliza el comando "RUN", se producirá una reinicialización del ordenador.

Un ensamblador permite utilizar lo que se conoce por *etiquetas* para realizar llamadas a las distintas partes de un programa en código de máquina, en lugar de hacer las llamadas directamente a las posiciones de memoria. Se trata de una de las funciones más importantes de los ensambladores y permite hacer las llamadas de manera similar al Pascal. (Pascal en un lenguaje de alto nivel, como lo es BASIC, pero sus programas no son ejecutados hasta haber sido ensamblados. Los programas objeto que se crean con este lenguaje no son tan rápidos como los que se programan en lenguaje ensamblador, y ocupan más espacio, pero son mucho más rápidos que los de BASIC).

En lugar de llamar a las subrutinas con GOSUB seguido de un número de línea, lo que se hace en Pascal es dar un nombre a cada subrutina. Este nora-



bre se puede colocar en el programa y, cuando se la encuentra, se ejecuta la subrutina. El ensamblador permite poner una etiqueta (que será un nombre seguido del símbolo ':') al lado de una instrucción; para llamar dicha instrucción se utiliza entonces la etiqueta. Es como si en BASIC se pudiera utilizar GOSUB seguido del nombre de la subrutina, sin necesidad de especificar en qué línea comienza ésta.

El ensamblador utiliza también *seudo-operaciones*; se las escribe de manera semejante a las operaciones normales del Z80, pero su efecto es diferente. Las principales son:

;	Hace que el resto de la línea sea considerado un comentario (como el REM de BASIC); el ensamblador ignora lo que sigue al punto y coma.
EQU	de <i>EQUate</i> o <i>EQUals</i> . Sirve para representar un número por una etiqueta. Primero se escribe la etiqueta, seguida de los dos puntos; a continuación se pone EQU y luego el número. Si se utiliza por ejemplo ETIQ: EQU #1234, entonces la etiqueta ETIQ se interpretará como el número 1234h (4660 decimal) cada vez que aparezca.
DEFB	de <i>DEFine Byte</i> . Define el contenido de un byte. El byte que corresponda a la instrucción será cargado con el valor que sigue a DEFB. Por ejemplo DEFB #20 cargará el número 20h en el byte que corresponda al ensamblar el programa.
DEFW	de <i>DEFine Word</i> . Es como la anterior, pero carga un número de 16 bits en dos posiciones sucesivas de memoria.
DEFM	de <i>DEFine Message</i> . Coloca los códigos ASCII del mensaje entrecomillado que se escriba después de DEFM en posiciones sucesivas de memoria.
DEFS	de <i>DEFine Space</i> . El ensamblador dejará en blanco tantas posiciones de memoria como indique el número que sigue a DEFS.
ORG	de <i>ORiGinate</i> . El número que sigue a ORG será la dirección que se dará a la instrucción siguiente al ensamblar el programa.
ENT	de <i>ENTry</i> . El número que sigue a ENT indica la dirección en que comenzará la ejecución del programa objeto cuando se utilice el comando J del ensamblador.

El programa CARGADOR HEX (que se encuentra en el apéndice B y del que hablaremos más adelante) necesitará que le proporcionemos la dirección inicial de una sección de programa; la encontraremos en los listados a continuación de ORG.

Cuando se desee ejecutar un programa desde BASIC se deberá llamar con CALL a la posición en que debe arrancar el programa; en los listados, esta dirección figura a continuación de ENT.

## Listados de ensamblador

Los listados de los programas que proporciona el ensamblador se componen de 5 columnas, o de 6 cuando se utilizan comentarios (que irán precedidos de ';').

La primera columna contiene las direcciones en que comienzan las instrucciones. Habitualmente la dirección figura en forma hexadecimal.

La segunda proporciona la versión hexadecimal de la instrucción de código de máquina, correspondiendo cada byte a dos cifras hexadecimales. Para cargar un programa con el CARGADOR HEX del apéndice B, ésta será la versión que tendremos que utilizar.

La tercera es un número de línea y no se utiliza más que al escribir el programa.

La cuarta columna la ocupan las etiquetas. En el listado no figuran los dos puntos que deben colocarse detrás del nombre de la etiqueta al escribir el programa. Si se copia un programa de un listado hay que acordarse de colocar los dos puntos detrás de cada etiqueta.

La quinta está ocupada por el código nemotécnico de la operación, tal como se escribe cuando se utiliza un ensamblador.

En la sexta columna puede aparecer un comentario.

Tras esta información básica, puede usted continuar la lectura.

## 4

### Diagramas de flujo

Como ayuda para el diseño y el desarrollo de un programa se utilizan a veces *diagramas de flujo*, que son esquemas simbólicos de las distintas partes del programa. Existe una serie de símbolos, con significado estándar, que se utilizan para realizar estos diagramas. Los más utilizados son los que se muestran en la figura 4.1.

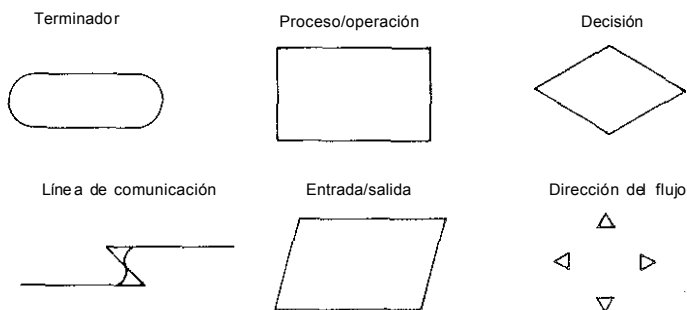


Figura 4.1

Hay muchos otros símbolos, pero son menos utilizados.

Los diagramas de flujo sirven para aclarar la secuencia de operaciones que realiza el programa. En la preparación de muchos programas es casi imprescindible comenzar por realizar el diagrama de flujo, para analizar las diferentes acciones que se deben realizar. También ayuda a prevenir los fallos antes de que ocurran, ya que permiten abarcar todo el programa de un vistazo.

Como ejemplo, la figura 4.2 nos muestra el diagrama de la operación que consiste en cargar en el ordenador un programa grabado en cinta.

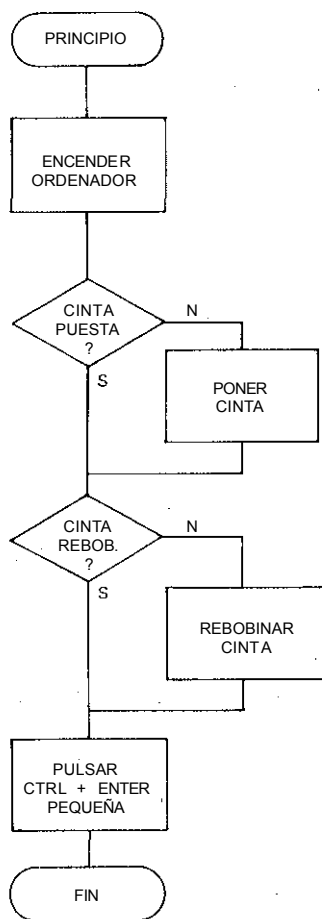


Figura 4.2

Los diagramas de la figura 4.3 ilustran la diferencia entre los bucles WHILE y los bucles FOR NEXT de BASIC. La diferencia entre ambos es evidente.

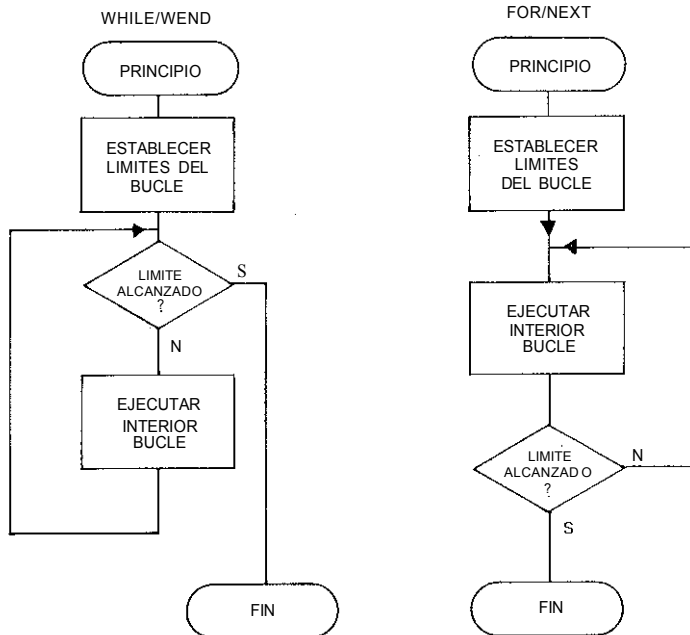


Figura 4.3



## Primeras instrucciones en código de máquina

### Instrucciones de carga

El Z80 tiene 14 *registros*, en los que se almacenan valores de manera similar a como lo hacen las variables enteras en BASIC. La figura siguiente representa esquemáticamente estos registros y la función que realizan. No se preocupe si hay muchas cosas que no entiende; el objetivo de este libro es precisamente aclarárselas.

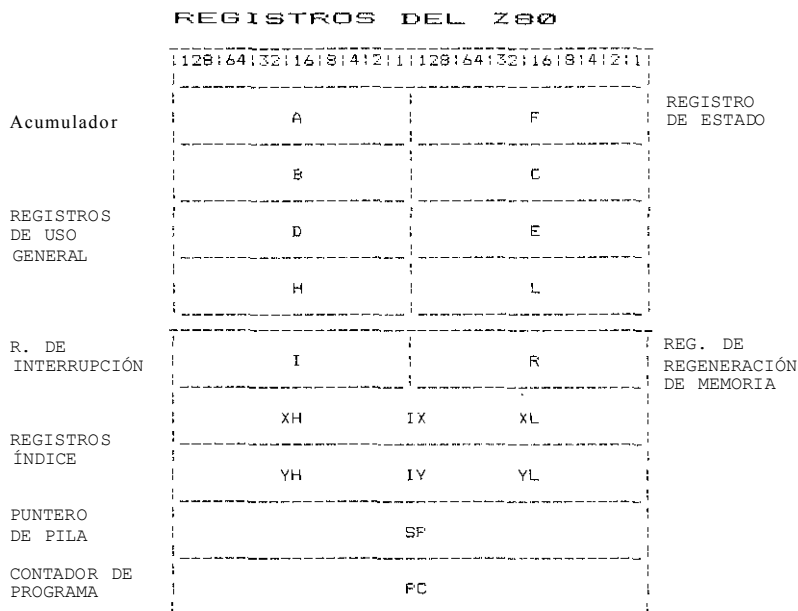


Figura 5.1

En este capítulo vamos a utilizar los seis registros de uso general: B, C, D, E, H y L; usaremos también el registro acumulador, A, y el registro contador de programa, PC, que se utilizan para tareas especiales.

El acumulador y los registros de uso general pueden almacenar un número comprendido entre 0 y 255, están formados por 8 bits y pueden ser cargados de tres maneras diferentes. Para entender las formas en que se pueden cargar estos registros, continuando con la analogía entre un registro y una variable de BASIC, escriba el programa de la figura 5.2. No es necesario que borre el primer programa si está todavía en la memoria.

```

180 CLS
190 WINDOW#1, 1, 40, 1, 10
200 WINDOW#2, 1, 40, 13, 23
210 WINDOW#3, 1, 40, 12, 12
220 PEN#3, 2: PRINT#3, " DECIMAL    BINAR
    IO    HEX"
230 INPUT#1, "INTRODUZCA UN NUMERO ";A
240 IF A > 255 THEN PRINT#1, "NUMERO NO
    VALIDO, DEBE SER MENOR DE 256": GOTO 230
250 A = INT (A)
260 PRINT#2,USING "#####"; A; : PRINT#2
    , " "; BIN$(A,8); " "; HEX$ (A,2)
270 PRINT#1 : PRINT#2
280 GOTO 230

```

Figura 5.2

Al ejecutar el programa con RUN 180, se le pedirá que introduzca un número. La variable A del programa representa el acumulador. Al introducir un número éste se carga en A, donde queda almacenado para su posterior utilización en otras tareas del programa. En este caso, si el número está entre 0 y 255, aparecerá en la pantalla en tres formas: decimal (que es como se lo ha introducido), binaria (que es como lo almacena el ordenador) y hexadecimal. Si, por ejemplo, el número introducido es 77, se cargará en A el valor 77.

La instrucción en código de máquina que permite cargar 77 en el acumulador es 'LD A,77', que es bastante fácil de recordar. 'A' es el símbolo del acumulador y 'LD' es la abreviatura de *load*, que es *cargar* en inglés. En realidad, LD A,77 no es una instrucción que entienda el ordenador directamente. Lo que el ordenador necesita es 00111110 seguido de 01001101, o bien 3Eh seguido de 4Dh, o 62 y 77 en decimal. Pero, si tenemos un ensamblador, podremos escribir LD A,77 y el ensamblador se encargará de traducirlo. LD A,77 es el código nemotécnico de la operación.



Volvamos ahora a la línea 90 del programa del capítulo 2. Era una sentencia DATA y el tercer dato era 62, el código de la instrucción para cargar el acumulador. El dalo siguiente era 65, el código ASCII de la 'A', que era la letra que queríamos escribir 255 veces. Justamente, 255 es el segundo dato de la línea. Pero el 6, ¿qué representa? El código de la operación que sirve para cargar el registro B con un número es 00000110 en binario o 6 en decimal y hexadecimal. Las dos primeras instrucciones del programa en código de máquina eran, pues,

LD B,255

LD A,65

No tendrá ahora dificultades para cambiar un poco aquel programa. Puede cambiar el número de veces que es escribe el carácter y también el carácter que se debe escribir.

Al cambiar el programa deberá suprimir o modificar la línea 40. Estaba pensada para comprobar, mediante el resultado de una suma, la exactitud de los datos de la línea 90. Si usted los cambia sin más, la suma le daría incorrecta.

Para cambiar el carácter que se escribe tendrá que consultar la tabla de códigos ASCII y encontrar el del carácter que desea; la tabla está en el apéndice III de la Guía del usuario de su Amstrad. Cambie el 65 por el código que desee, pero no utilice ningún valor inferior a 32, pues se trata de códigos de control y obtendría resultados inesperados.

Cambie también el 255 por el número de veces que desea que se imprima el carácter; este número no puede exceder de 255. Sin embargo, si reemplaza 255 por 0 encontrará que el carácter se escribe 256 veces; ¿por qué? La línea 60 del programa, que contiene en BASIC el proceso análogo al que realiza la rutina de código de máquina, puede darnos la explicación. El registro B contiene 0 y en el primer paso se cambia este valor por B-1=0-1. Ahora bien, la operación en binario da 00000000b-00000001b=11111111B, que es 255. La misma respuesta le dará el ordenador si usted escribe '?BIN\$(-1)'. ¿Le parece confuso?; repase entonces el capítulo 3 de este libro o el apéndice II de la Guía del usuario.

Todos los registros de uso general pueden ser cargados con un número de 8 bits de la misma manera que A y B. Los códigos de las operaciones son los que se muestran en la figura 5.3. En todos los casos, n representa el número, entre 0 y 255 decimal (FFh y 11111111b), que se debe cargar en el registro.

Si observa atentamente el código binario debe notar dos cosas. Lo primero que comienza y termina igual en todos los casos. Estas dos partes son las que indican al microprocesador que debe cargar un número en un registro.

En segundo lugar, el registro que se carga viene indicado por los bits 5,

ENSAMBLADOR	DECIMAL	HEX	BINARIO
LD B,n	06 n	06 n	00 000 110 n
LD C,n	14 n	0E n	00 001 110 n
LD D,n	22 n	16 n	00 010 110 n
LD E,n	30 n	1E n	00 011 110 n
LD H,n	38 n	26 n	00 100 110 n
LD L,n	46 n	2E n	00 101 110 n
LD A,n	62 n	3E n	00 111 110 n

Figura 5.3

4 y 3. Siempre que una operación concierne a uno de los registros de uso general se utilizan estas mismas combinaciones de 3 bits para decirle de qué registro se trata. Así pues,

B es siempre 000  
 C es siempre 001  
 D es siempre 010  
 E es siempre 011  
 H es siempre 100  
 L es siempre 101  
 A es siempre 111

Figura 5.4

De las 8 posibles combinaciones de 3 bits falta la 110; ésta se utiliza para un objetivo especial que explicaremos en este mismo capítulo.

De la misma manera que es posible cargar un registro directamente con un número, también es posible hacerlo indirectamente con el contenido de otro registro o con el contenido de una posición de memoria.

Piense en la sentencia de BASIC A=B. Lo que hace es cargar en la variable A el mismo valor que hay cargado en la variable B. Esto, sin embargo,

no cambia el valor que haya en B. Puede comprobarlo escribiendo las líneas de la figura 5.5 y ejecutándolas con RUN 300; tras la línea 320, A tendrá el mismo valor que B, pero B no habrá cambiado.

```

300 B = 10

310 PRINT " ANTES: A=";A;" B=";B

320 A = B

330 PRINT "DESPUES: A=";A;" B=";B

```

Figura 5.5

Sabiendo que el código de máquina nemotécnico equivalente a la sentencia de 300 es LD B,10, ¿cual será el equivalente a la sentencia de la línea 320? No es difícil imaginar que es LD A,B- De manera similar se obtienen todas las instrucciones de carga de un registro en otro.

En lo que se refiere al código binario de estas instrucciones, se forma de manera parecida al de la carga de un registro con un número. Los bits 7 y 6 son ahora 01 en lugar de 00; los siguientes 3 bits son el identificador del registro de destino; finalmente, los 3 últimos se completan con el código del registro de origen en lugar del fijo 110. Así, tenemos ahora

ENSAMBLADOR	DECIMAL	HEX	BINARIO
LD A,B	120	78	01 111 000

Recuérdelo, el código para cargar un registro en otro tiene fijos los bits 7 y 6 con 01, los 3 bits siguientes representan al registro de destino y los 3 últimos al registro origen. Puede usted ejercitarse en encontrar los códigos de las diferentes posibilidades.

Ya conocemos dos formas de cargar registros. Habrá observado que la forma de construir las instrucciones es completamente lógica. Si lo ha entendido así no tendrá dificultades para seguir.

Todos los registros de uso general poseen aspectos específicos que serán examinados a lo largo del libro. Lamentablemente, y en esto se diferencian mucho de las variables de BASIC, no está en la mano del usuario decidir las limitaciones que posee cada registro. Cuando se enciende el ordenador, cualquier variable puede servir para cualquier cosa; por el contrario, sólo ciertos registros pueden servir para determinadas tareas.

Esto puede entenderse mejor con ayuda de un ejemplo. Añada usted al programa del capítulo 2 la línea '21 DEFSTR A' y ejecute el programa. Ob-

tendrá un mensaje de error que se debe a la utilización como variable numérica de una variable que sólo puede ser una cadena literal.

Observemos asimismo la diferencia que existe entre las instrucciones de BASIC '?' y '?PEEK(8)'. La respuesta a la primera será 8, mientras que la segunda imprimirá 195. No es lo mismo preguntar "qué es 8" que preguntar "qué hay en la posición 8 de la memoria".

Pues bien, también es posible cargar en un registro "el contenido" de una posición de memoria. Pero ahora el acumulador A es el único registro de 8 bits que se puede cargar de esta manera. Vamos a explicar el equivalente a la instrucción de BASIC

A=PEEK(nn)

donde nn es un número de 16 bits.

Si se desea cargar el acumulador A con el contenido de la posición de memoria número 8, la instrucción nemotécnica no puede ser LD A,8, pues ésta cargaría en el acumulador el número 8. Para indicar que se trata del *contenido* de la posición 8 (y no del número 8) se emplea el paréntesis, como en PEEK(8), y se escribe LD A,(8). O sea, (nn) significa "el contenido de nn".

También se puede realizar la operación contraria, equivalente a POKE nn,A, para cargar una posición de memoria con el contenido del acumulador A. Su código nemotécnico es LD (nn),A. Por ejemplo, la instrucción LD (40000),A sirve para cargar en la posición de memoria 40000 el contenido de A.

Si no se dispone de ensamblador, las cosas se complican un poco más, aunque no demasiado. Los códigos son

ENSAMBLADOR	DECIMAL	HEX	BINARIO
LD A, (nn)	58 n n	3A n n	00 111 010 n n
LD (nn),A	50 n n	32 n n	00 110 010 n n

El número nn representa una dirección de memoria y es de 16 bits, es decir, ocupa dos posiciones de memoria. Es fundamental saber y recordar que para el número nn cada una de las dos n se debe calcular mediante la fórmula:

$n1 = \text{número} \text{ MOD } 256 \quad \text{y} \quad n2 = \text{INT}(\text{número}/256)$

Puede parecer sorprendente que, de los dos bytes que componen el número nn, el menos significativo se deba colocar primero y el más significativo el segundo. El Z80 trabaja siempre de esta manera con los números de 16 bits, tanto para cargarlos como para almacenarlos en memoria.

Es fácil escribir un pequeño programa que calcule para cada número de 16 bits los números  $n_1$  y  $n_2$ . Pero no conviene utilizar la función MOD del ordenador ya que, al utilizarse a veces la representación normal de un entero y otras la notación en complemento a 2, resulta desaconsejable para números mayores de 32767. Es mejor utilizar nuestra propia fórmula y escribir

```
1010 N2 = INT(NUMERO/256) : N1 = NUMERO
- N2 * 256 : PRINT "N1 =";N1;" N2 =";N2
```

Si ahora ejecutamos esta línea con

```
NUMERO=40000: GOTO 1010
```

obtendremos  $N_1=64$   $N_2=156$  como respuesta. Con estos números podemos construir los códigos completos de carga y descarga de la posición 40000,

ENSAMBLADOR	DECIMAL	HEX
LD A, (40000)	5B 64 156	3A 40 9C
LD (40000), A	50 64 156	32 40 9C

BINARIO							
LD A, (40000)	00	111	010	0100	0000	1001	1100
LD (40000), A	00	110	010	0100	0000	1001	1100

y, análogamente, los de la posición 8,

ENSAMBLADOR	DECIMAL	HEX
LD A, (8)	58 8 0	3A 08 00
LD (8), A	50 8 0	32 08 00

BINARIO							
LD A, (8)	00	111	010	0000	1000	0000	0000
LD (8), A	00	110	010	0000	1000	0000	0000

Puede practicar con lo que acabamos de ver cambiando el programa del capítulo segundo. En aquel programa se utilizaba la instrucción LD A,65, que ahora podemos sustituir por LD A,(8), por ejemplo. Para ello la línea 90 se debe sustituir por

```
90 DATA 6,255,58,8,0,205,90,187,16,251,201
```

cambiando en consecuencia la cantidad de la línea 40 que se emplea en la comprobación por 1277.

Es importante cambiar la línea 30 por

```
30 FOR N=43800 TO 43810:READ D:POKE N,D:A=A+D:NEXT
```

ya que hay un byte más en la rutina.

Para hacer que la rutina BASIC de la línea 60 corresponda a la nueva rutina en código de máquina, la línea 60 se debe sustituir por

```
60 PRINT CHR$(PEEK(8));: B=B-1 : IF B<>0 THEN 60
```

Finalmente, se debe borrar la línea '21 DEFSTR A' si fue introducida. Al ejecutar ahora el programa se obtendrá el símbolo que corresponde al código 195 (una barra \) en lugar de la A.

Pasaremos a continuación a explicar otra forma de utilización de los registros de uso general. Los registros de uso general pueden ser utilizados agrupados en los pares BC, DE y HL, constituyendo así 3 registros de 16 bits. Esto permite utilizar registros que pueden cargar números comprendidos entre 0 y 65535, en lugar de entre 0 y 255 como antes.

Una diferencia con respecto de la utilización individual de los registros es que no hay ninguna instrucción del tipo LD rr,rr', que permita cargar un par de registros con el contenido de otro par. Por el contrario, existen otras diferencias en sentido positivo.

Para cargar en un par de registros rr un número nn de 16 bits, el código nemotécnico es LD rr,nn (rr representa BC, DE o HL). Así, las instrucciones

```
LD BC,40000
LD HL,8
```

sirven para cargar 40000 en el par BC y 8 en el par HL, respectivamente.

La construcción de los códigos binarios es similar a la de los códigos de las instrucciones LD r,n. Los 2 primeros bits son 00 como en aquel caso. Después vienen 2 bits que indican el par que se carga; son simplemente los mismos cuando actúa un par de registros:

00 es siempre el par BC  
 01 es siempre el par DE  
 10 es siempre el par HL

Luego viene un 0 y finalmente los 3 bits 001. Tenemos así

ENSAMBLADOR	DECIMAL	HEX	BINARIO
LD BC,nn	1 n n	01 n n	00 000 001 n n
LD DE,nn	17 n n	11 n n	00 010 001 n n
LD HL,nn	33 n n	21 n n	00 100 001 n n

El código del número nn ocupa 2 bytes y se obtiene como indicamos anteriormente (primero el byte menos significativo). Por ejemplo,

ENSAMBLADOR	DECIMAL	HEX	BINARIO
LD BC,40000	1 64 156	01 40 9C	00 000 001 0100 0000 1001 1100
LD HL,8	33 8 0	21 08 00	00 100 001 0000 1000 0000 0000

Como los pares de registros cargan números de 16 bits y éste es también el tamaño de las direcciones de memoria, se los utiliza particularmente para apuntar a posiciones de la memoria. Ya hemos dicho que no existen las instrucciones LD r,(nn) ni LD (nn),r cuando r es un registro de uso general; pero hay una forma de suplir esa carencia. Se trata de apuntar a la dirección cuyo contenido se desea cargar (o viceversa) con el par HL. Todo ocurre como si en BASIC estuviese prohibido utilizar 'B=PEEK (8)', pero se pudiese hacer 'B=PEEK(HL)' dando a HL el valor 8.

Se puede cargar cualquier registro de uso general, y también A, con el contenido de la memoria a la que apunta el par HL. También se puede cargar la posición de memoria a la que apunta HL con el contenido del acumulador o de un registro de uso general. Estos dos tipos de instrucciones tienen códigos nemotécnicos de la forma LD r,(HL) y LD (HL),r. Aquí representa A, B, C, D, E, H o L. Los paréntesis que rodean HL significan que se trata del contenido de una posición de memoria (y no del contenido de HL).

Sus códigos binarios completan el vacío que existía en los códigos que comenzaban por 01; interviene aquí justamente el código de 3 bits 110b, que no representaba ningún registro. Los códigos de ambas comienzan por 01. Los de LD r,(HL) son de la forma

[01] [código de 3 bits del registro] [110]

y los de LD (HL),r, de la forma

[01] [110] [código de 3 bits del registro]

Por ejemplo, los códigos

ENSAMBLADOR	DECIMAL	HEX	BINARIO
LD HL,40000	33 64 156	21 40 9C	00 100 001 0100 0000 1001 1100
LD D, (HL)	86	56	01 010 110

sirven para cargar en D el contenido de la posición 40000 de memoria, y los códigos

ENSAMBLADOR	DECIMAL	HEX	BINARIO
LD HL,8	33 8 0	21 08 00	00 100 001 0000 1000 0000 0000
LD B, (HL)	70	46 01	000 110

para cargar en B el contenido de la dirección 8 de memoria.

Para probar sus nuevos conocimientos puede cambiar la rutina ya utilizada, sustituyendo la línea 90 por

```
90 DATA 33,8,0,70,58,8,0,205,90,187,16,251,201
```

y realizando en el programa los cambios necesarios. La suma de comprobación de la línea 40 será ahora 1127; en la línea 30 el límite superior del bucle debe ser 43812. Finalmente, para que la rutina BASIC de la línea 60 coincida con la de código de máquina, deberá poner HL=8 : B=PEEK (HL ) en lugar de B=255 en la línea 50. El comienzo de la rutina en código de máquina es ahora

ENSAMBLADOR	DECIMAL
LD HL,8	33 8 0
LD B,(HL)	70
LD A,(8)	58 8 0

El código 110b significa, pues, (HL) cuando se lo coloca en la posición que debe ocupar un registro. En otros casos tiene un significado diferente; así, una instrucción del tipo LD r,n cuyo código es '[00] [código de r] [110]', el 110 significa que el siguiente byte debe ser interpretado como un número. Pero en esta misma instrucción, si 110b se coloca en el lugar de r formando el código 00 110 110, este código es el de la instrucción LD (HL),n, cuya



finalidad es colocar el número  $n$  en la posición de memoria a la que apunta HL.

Observe finalmente que la sustitución en la instrucción LD  $r, r'$  de ambos registros por (HL) carece de sentido y que por lo tanto el código 01 110 110 no tendrá el significado de una instrucción de carga. De hecho, este código posee un sentido completamente diferente: su efecto es detener el Z80.

Cuando se utiliza el acumulador, las instrucciones de carga relativas a una posición apuntada por un par de registros pueden usar como puntero, no sólo el par HL, sino también los pares BC y DE. Es decir, son válidas las instrucciones LD A,(rr) y LD (rr),A cuando rr es cualquiera de los pares BC, DE y HL, lo que nos da las nuevas instrucciones

LD DE,8  
LD A,(DE)

El diseño de los códigos binarios de estas operaciones difiere del de las instrucciones LD A,(HL) y LD (HL),A, que empezaban por 01 (recuerde que las posibilidades de comenzar por 01 están agotadas). Lo que hacen es seguir el modelo de las instrucciones LD A,(nn) y LD (nn),A.

Los códigos que representan pares de registros y los que representan un registro están relacionados de forma sencilla: el código 00 representa el par BC, y los códigos para B y C son 000 y 001, es decir, comienzan con 00. Lo mismo ocurre con DE, D y E (01, 010 y 011) y con HL, H y L (10, 100 y 101).

El código de LD A,(nn) es 00 111 010. En las instrucciones de carga de un par, que también comenzaban por 00, los bits 5 y 4 representaban el código del par. Aquí esos bits contienen 11, que es el único código de dos bits que no estaba asignado. Esto explica que el código de LD A,(BC) sea 00 001 010 y el de LD A,(DE) sea 00 011 010.

Siguiendo esta lógica, 00 101 010 debería ser el código de LD A,(HL), pero ya sabemos que no es así; pronto diremos a qué corresponde este código.

El código de LD A,(nn) es 00 110 010; la misma lógica que antes lleva a que el código de LD (BC),A sea 00 000 010 y el de LD (DE),A 00 010 010. Tampoco en este caso 00 100 010 es el código de LD (HL),A.

Estas instrucciones de carga se refieren a cantidades que ocupan un byte. Las que vamos a ver a continuación transfieren cantidades que ocupan dos bytes. Además vamos a encontrar un dueño para los dos códigos que no lo tenían.

Se trata de las instrucciones LD HL,(nn) y LD (nn),HL, que funcionan de manera similar a LD A,(nn) y LD (nn),A, es decir, cargando el contenido del registro en una posición de memoria o viceversa. En primer lugar está la cuestión del código binario. Este código consta de un byte con el código de operación y dos bytes con el número de 16 bits nn, que indica una posi-

ción de memoria. Ya hemos explicado cómo se obtienen las dos partes n1 y n2 del número nn. Por ejemplo, para nn=8 se obtiene n1 0000 1000 y n2 0000 0000.

Los códigos de operación son justamente

ENSAMBLADOR	BINARIO
LD HL, (nn)	00 101 010 n n
LD (nn), HL	00 100 010 n n

o sea, los que habíamos echado en falta.

Hay sin embargo una dificultad que hemos eludido: HL almacena cantidades de 16 bits, mientras que la capacidad de una posición de memoria es de solamente de 8 bits. ¿Cómo se produce entonces la transferencia? Lo que ocurre es que no se utiliza una posición de memoria, sino dos. La posición de memoria nn efectúa la transferencia con L y la posición siguiente (nn+1) con H; es decir, se tiene el esquema

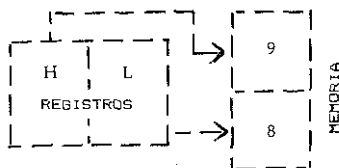


Figura 5.6

Las palabras inglesas *low* (bajo) y *high* (alto) justifican las denominaciones L y H y una manera cómoda de recordar cómo se efectúan las transferencias con los pares de registros. De hecho, esta misma técnica se emplea para cualquier par de registros; el registro que se escribe a la derecha (L, C o E, según el caso) se corresponde con la primera de dos posiciones de memoria y con el byte menos significativo; el registro que se escribe a la izquierda (H, B o D) se corresponde con la segunda de dos posiciones de memoria y con el byte más significativo.

Las últimas instrucciones de carga que vamos a ver son similares, pero utilizan los pares BC y DE. Son

LD BC,(nn) LD DE,(nn) LD (nn),BC y LD (nn),DE

Se las utiliza con menor frecuencia que las de HL porque su código ocupa más espacio; exactamente 4 bytes, dos para el código y dos para nn. Los códigos son

ENSAMBLADOR	DECIMAL	HEX	BINARIO
LD BC, (nn)	237 75 n n	ED 4B n n	1110 1101 01 001 011 n n
LD DE, (nn)	237 91 n n	ED 5B n n	1110 1101 01 011 011 n n
LD (nn),BC	237 67 n n	ED 43 n n	1110 1101 01 000 011 n n
LD (nn),DE	237 83 n n	ED 53 n n	1110 1101 01 010 011 n n

Figura 5.7

Se observará que todos ellos comienzan por el hexadecimal ED (1110 1101b o 237 decimal, pero el hexadecimal es más fácil de recordar). El prefijo ED es el que sirve para alterar el significado del segundo byte, lo que debe recordarle algunas consideraciones sobre el lenguaje que hicimos en el capítulo 2.

Al final de este capítulo incluimos un pequeño resumen de las instrucciones que comienzan por LD. También encontrará una descripción más gráfica y detallada en el apéndice A.

### Llamadas. El contador de programa (PC)

Siempre que el ordenador está encendido, y salvo que el microprocesador esté detenido por alguna razón, el registro *contador de programa* (que se denota por PC como consecuencia de su nombre en inglés, que es *Program Counter*) se ocupa de controlar las operaciones del Z80. Actúa como si su finalidad consistiese en aumentar su valor hasta llegar al final de la memoria y recomenzar nuevamente. El valor almacenado en PC es el de la dirección de memoria de la instrucción que el microprocesador debe ejecutar. Al encender el ordenador el valor que se carga en PC es 0; por lo tanto debe estar ahí la primera instrucción a ejecutar. Lo que el ordenador hace entonces es ejecutar un programa que lo coloca a disposición del usuario, en modo BASIC para el caso del Amstrad. Este programa inicial recibe el nombre de *arranque en frío*.

En todo momento el microprocesador está ejecutando algún programa y, lógicamente, es esencial tener un control sobre su evolución, o sea, sobre el contador de programa. Si el microprocesador ejecutase linealmente las ins-

trucciones que hay en la memoria, el ordenador tendría la misma utilidad que un piano en el que sólo se pudiesen tocar las teclas desde la primera a la última. Cambiando la afinación del piano llegaríamos a tocar alguna melodía, pero, a continuación, sólo podríamos hacer que la repitiese sin parar.

Por suerte es posible cambiar el orden en que el microprocesador ejecuta las instrucciones. Las instrucciones de BASIC que alteran el orden de ejecución de las líneas de un programa BASIC son GOTO, GOSUB y RETURN, GOTO hace que se salte a la línea indicada; GOSUB hace saltar a una subrutina y RETURN termina la subrutina y devuelve el control al programa principal. En código de máquina existen las instrucciones equivalentes a éstas. Las que equivalen a GOTO tienen como códigos nemotécnicos JP y JR, que provienen de la palabra *jump* (salto). Las que se asemejan a GOSUB y RETURN tienen los códigos CALL (llamar) y RET (*return*, volver).

CALL y RETURN funcionan como sus equivalentes de BASIC. La instrucción CALL debe ir acompañada de la posición de memoria de la primera instrucción de la subrutina (es lo que equivale al número de línea de BASIC). Esta dirección de memoria ocupa 2 bytes y se carga en la forma habitual de menos significativo y más significativo; el cálculo de estos 2 bytes se realiza como ya hemos explicado. La instrucción CALL ocupa pues 3 bytes; uno para el código y dos para la dirección. Los códigos de ambas instrucciones son:

ENSAMBLADOR	DECIMAL	HEX	BINARIO
CALL nn	205 n n	CD n n	11 001 101 n n
RET	201	C9	11 001 001

Vuelva al programa BASIC del capítulo 2, cuya línea 90 contiene los datos de un programa en código de máquina. Detrás de los valores con los que ha experimentado anteriormente encontrará 205,90,187; se trata de una instrucción CALL. El primer número es el código de CALL y los dos siguientes proporcionan la dirección de la instrucción que se llama. Ya sabemos descifrar esta dirección; hay que sumar al segundo número el tercero multiplicado por 256:

$$187 \times 256 = 47872. \quad 47872 + 90 = 47962 \text{ o BB5Ah}$$

El programa en código de máquina comienza, pues, por cargar en el registro A el código del carácter que se debe escribir, y en el registro B el número de veces que va a ser escrito; a continuación llama a la subrutina que comienza en la dirección 47962(BB5Ah). Esta subrutina es parte del sistema operativo; es probablemente la subrutina del sistema operativo que deberá utilizar con mayor frecuencia. Amsoft le ha dado el nombre de TXT OUTPUT y

lo que hace es escribir el carácter cuyo código se encuentra en el acumulador en la ventana de pantalla que se esté utilizando en la actualidad.

Esta subrutina entiende también los códigos de control que se explican en el capítulo 9 de la Guía del usuario. Para ver cómo responde a los códigos de control, cambie la línea 90 del programa por

```
90 DATA 62,7,205,90,187.201
```

el número que sigue a TO en la línea 30 por 43805 y la suma de comprobación de la línea 40 por 752. El programa que se carga así es, en ensamblador,

```
LD A.7
CALL 47962
RET
```

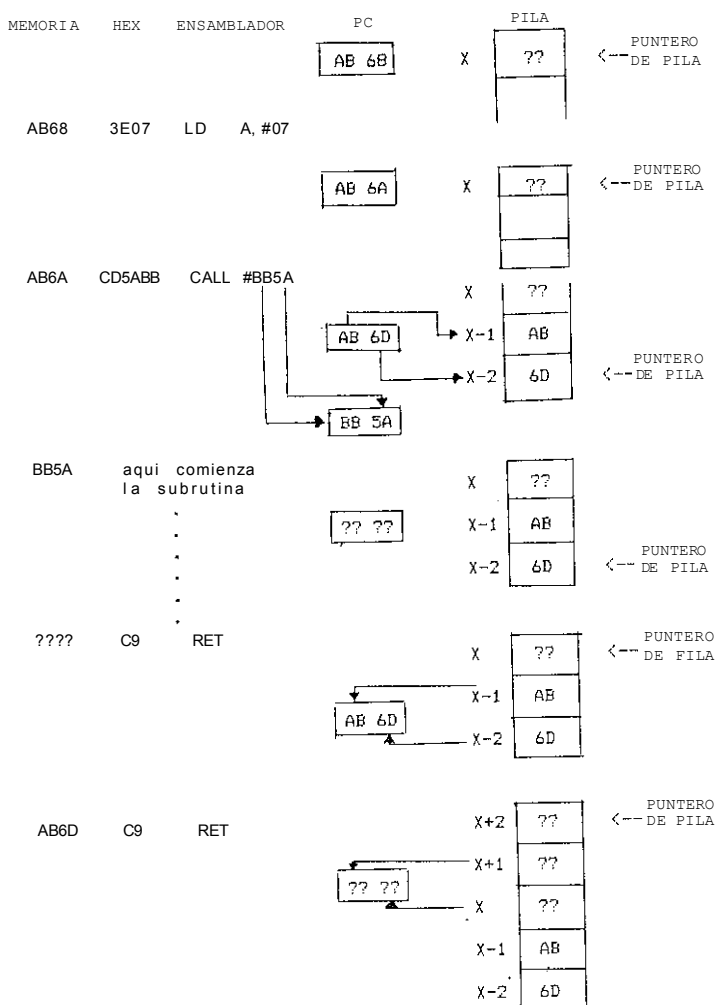
Al ejecutarlo debe usted oír un pitido. Si no es así, vuelva a intentarlo tecleando directamente CALL 43800 seguido de la tecla [ENTER]; de esta manera estará llamando directamente al programa en código de máquina sin necesidad de volver a ejecutar el programa en BASIC.

Al contrario de lo que ocurría en las instrucciones LD, aquí no es posible dar la dirección de llamada como la dirección a la que apunta un par de registros. La instrucción CALL debe ir seguida de 2 bytes que representen la dirección explícita.

Cuando al final de la subrutina se ejecuta la instrucción RET, el control pasará a la posición de memoria que sigue a los 3 bytes ocupados por la instrucción CALL. Para poder hacer esto el microprocesador debe recordar dónde estaba situada la instrucción CALL. Esto es posible mediante la utilización de la *pila*, que es un pequeño archivo que utiliza el Z80. Vamos a ver cómo se utiliza la pila en el caso de las instrucciones CALL y RET, dejando para el capítulo 9 una descripción más detallada de la utilización de la pila.

Para imaginarse el funcionamiento de la pila viene bien compararla con un clavo situado en el techo en el que los bytes de información se almacenan como se haría con trozos de papel que se pinchasen en el clavo. A medida que un dato se introduce en la pila, ésta crece hacia abajo. Por otra parte, la información de la pila sólo puede recuperarse a partir de la que está situada más abajo, que es la última que se ha introducido.

La pila ocupa cierto área de la memoria. La posición de memoria más baja ocupada por la pila está siempre almacenada en el registro puntero de pila, que se denota por SP (del inglés *Stack Pointer*). Hay que preocuparse de que el programa no modifique involuntariamente la zona de memoria ocupada por la pila; de otra manera, el programa fallaría con toda seguridad. Lo mejor es situar la pila en lo alto de una gran zona libre de la memoria, lo que permitirá que la pila crezca hacia abajo sin topar con otra cosa.



Cuando el programa llega a una instrucción CALL, el microprocesador coloca la dirección que está en el PC (la dirección de la instrucción que sigue a CALL) en la pila y carga en el PC la dirección de la subrutina. De esta manera, la siguiente instrucción que se ejecuta es la primera de la subrutina. Al final de la subrutina, cuando se llega a una instrucción RET, el microprocesador recupera de la pila la dirección a la que debe volver y la coloca en el PC, de manera que la instrucción que se ejecute a continuación sea la que sigue a la instrucción CALL.

La secuencia de la figura 5.8 muestra esquemáticamente lo que sucede cuando se ejecutan las instrucciones CALL y RET. La primera columna de la figura contiene las direcciones (en hexadecimal) de comienzo de las instrucciones; la segunda contiene el código hexadecimal de cada instrucción y la tercera los códigos nemotécnicos de las instrucciones, con los datos numéricos en versión hexadecimal. Si se ha utilizado la forma hexadecimal es para mostrar más claramente lo que sucede, ya que así cada dos cifras hexadecimales corresponden a un byte de memoria. El ejemplo que se utiliza es el programa de antes, cuyo efecto consistía en hacer sonar un pitido.

La mecánica es simple: al colocar un dato en la pila, ésta crece en 2 bytes; al recuperar un dato, la pila decrece en 2 bytes. La precaución fundamental que hay que tener al manejar la pila es no introducir ningún dato que no vaya a ser extraído posteriormente. Si no se tiene este cuidado se puede producir alguno de los dos errores siguientes: que la pila crezca demasiado, invadiendo el espacio reservado al programa, o que se recupere un dato que no es el que se desea. Más adelante veremos instrucciones que utilizan la pila y con las que hay que ser cuidadoso para respetar la regla fundamental de su manejo: la cantidad de información que entra debe coincidir con la que sale. El desequilibrio de la pila es la causa más frecuente de fracaso de los programas. Al contrario que en BASIC, aquí ocurre frecuentemente que lo único que se puede hacer cuando fracasa un programa es desconectar el ordenador y comenzar de nuevo.

## Saltos

Existen dos tipos de instrucciones de salto; el primero se asemeja totalmente a la sentencia GOTO de BASIC. La sentencia GOTO 100 tiene el efecto de saltar a una línea número 100 que debe existir en el programa. Como no existen números de línea en código de máquina, las instrucciones de salto transfieren el control a una dirección de memoria.

El código nemotécnico de este primer tipo de instrucciones es JP (abreviatura de *jump*, o sea, salto). Este código va seguido normalmente de 2 bytes con una dirección, que es la del salto. La forma del salto es semejante a la

de la instrucción CALL, pero ahora no está previsto ningún regreso y en consecuencia no se utiliza la pila. La forma completa de esta instrucción es JP nn; permite saltar a cualquier dirección de memoria accesible. Al igual que CALL, la instrucción se compone de 3 bytes, pero el primero, que era 11 001 101 para CALL, es ahora 11 000 011 para JP. Si en nuestro último programa utilizamos una instrucción JP en lugar de CALL, el código será

ENSAMBLADOR	DECIMAL	HEX	BINARIO
JP 47962	195 70 187	C3 5A BB	11 000 011 0101 1010 1011 1011

También es posible utilizar el par de registros HL para indicar la dirección del salto; en este caso el salto se realiza a la dirección contenida en el par HL. El código nemotécnico de esta instrucción es fácil de averiguar si se recuerda la notación de "contenido en". El código binario ocupa solamente un byte. Estos códigos son:

ENSAMBLADOR	DECIMAL	HEX	BINARIO
JP (HL)	233	E9	11 101 001

Los saltos están entre las instrucciones que más se utilizan. Modificados convenientemente y combinados con la instrucción CALL, permiten crear instrucciones análogas a las ON GOTO y ON GOSUB de BASIC; pero esto lo explicaremos en el próximo capítulo.

Vamos a explicar ahora el segundo tipo de instrucciones de salto. Muchos de los saltos necesarios se hacen a direcciones de memoria muy cercanas a la dirección en la que se está, que es la del PC. Puede resultar mejor ordenar un salto a 5 posiciones más adelante en lugar de explicitar la dirección del salto. Para ello existe la instrucción de salto relativo, cuyo código es JR (abreviatura de *jump relative*).

La instrucción JR se compone de 2 bytes. El primero contiene el código de operación y el segundo la magnitud del salto o, más exactamente, la distancia del salto desde la posición marcada por el PC (que es la de la instrucción siguiente) a la instrucción a la que se desea saltar. El salto puede ser hacia adelante o hacia atrás, siendo así su magnitud un número positivo o negativo. En la codificación en 1 byte de este número se emplea la notación del complemento a 2 que explicamos en el capítulo 3; esto hace posible que el número varíe entre +127 y -128. Los códigos de JR son:

ENSAMBLADOR	DECIMAL	HEX	BINARIO
JR n	24 n	18 n	00 011 000 n



La utilización de un ensamblador evita tener que calcular la magnitud de los saltos relativos, ya que se puede utilizar una etiqueta para marcar la posición a la que se debe saltar (el ensamblador se encargará de los cálculos). La etiqueta se puede definir colocándola en el programa o también mediante la pseudo-operación EQU que explicamos en el capítulo 3. Veamos dos ejemplos.

El programa del primer ejemplo tiene por efecto hacer sonar un pitido y escribir repetidamente la letra 'A':

	ETIQUETA	ENSAMBLADOR	DECIMAL	HEX
43880	{AB68h}	LD A, 7	62	3E 07
43882	{AB6Ah}	PRINT: CALL 47962	205 90 167	CD 5A BB
43885	{AB6Dh}	LD A, 65	62 65	3E 41
43887	{AB6Fh}	JR PRINT	24 249	18 F9

En este ejemplo, el 249 que hay después del código de operación 24 sirve para transferir la ejecución a la posición -7 en relación con el contenido del PC en ese momento, que será de 43889 ya que apunta a la siguiente instrucción. Como  $43889 - 7 = 43882$ , el salto se hará al comienzo de la instrucción CALL.

En el segundo ejemplo no sonará el pitido, ya que la instrucción LD A,7 no se ejecuta y lo primero que se escribe es la letra 'A':

	ETIQUETA	ENSAMBLADOR	DECIMAL	HEX
43880	{AB68h}	JR GO	24 5	18 05
43882	{AB6Ah}	LD A, 7	62 7	3E 07
43884	{AB6Ch}	PRINT: CALL 47962	205 90 187	CD 5A BB
43887	{AB6Fh}	GO: LD A, 65	62 65	3E 41
43889	{AB71h}	JR PRINT	24 249	18 F9

Nótese que aquí la instrucción JR GO tiene por efecto realizar un salto relativo de 5 posiciones a partir del contenido del PC, pues éste contendrá la dirección en la que comienza la instrucción LD A,7.

En general no habrá que efectuar cálculos cuando se utilice un ensamblador, salvo en el caso de programas muy largos, pues en ellos puede ser importante ahorrarse etiquetas para utilizar menos espacio.

Aquí hay que advertir que el ensamblador GENSA3 del paquete DEV-PAC de Highsoft complica singularmente la situación, ya que las distancias

de salto se calculan a partir del *contador de posición* del ensamblador y no del contenido del PC; esto es lo que se explica en la página 2.6 del manual del DEVPAC. El contador de posición, al que se hace referencia a través del símbolo \$, contiene la posición del comienzo de la instrucción JR cuando se llega a esta instrucción, luego hay que añadir 2 a la magnitud del salto si se la ha calculado de la forma habitual (a través de PC). Por ejemplo, si se desea suprimir la etiqueta en el anterior ejemplo, la instrucción JR PRINT se deberá escribir en la forma JR\$-5. en lugar del lógico JR - 7.

Por el contrario, no hay que preocuparse de esta diferencia si se utilizan etiquetas, ya que entonces el salto se realiza en cualquier caso a la dirección que señala la etiqueta.

Antes de terminar el capítulo veremos una última instrucción que es muy sencilla pero de gran utilidad; permite intercambiar entre sí los contenidos de los pares DE y HL, lo que resulta sumamente interesante cuando se tiene HL cargado con una dirección y se desea utilizarlo para cualquier otra cosa. Su código es EX DE,HL, donde EX se utiliza como abreviatura de *exchange* (intercambio). Los distintos códigos de la instrucción son:

ENSAMBLADOR	DECIMAL	HEX	BINARIO
EX DE,HL	235	EB	11 101 011

Por ejemplo, si DE está cargado con el número 10 y HL con 37, tras la ejecución de la instrucción el par DE contendrá 37 y HL contendrá 10.

## Resumen

Vamos a resumir las instrucciones explicadas en este capítulo. Utilizaremos los símbolos:

- r = cualquiera de los registros de 8 bits (A, B, C, D, E, H o L)
- rr = cualquier par de registros que se utilicen como uno de 16 bits
- n = un número de 8 bits, o sea, entre 0 255
- mn = un número de 16 bits, o sea, entre 0 y 65535
- ( ) rodeando un número o un par de registros=el contenido de la dirección.
- PC = contador de programa
- SP -puntero de pila

El código de las operaciones de carga es LD.

Todo r puede ser cargado con cualquier n; la instrucción tiene la forma **LD r,n.**

Todo *r* puede ser cargado con el contenido de cualquier otro *r*; la instrucción tiene la forma LD *r,r'*.

El registro *A* puede ser cargado con el contenido de una dirección de la memoria; la instrucción tiene la forma LD *A,(nn)*.

Una dirección de la memoria puede ser cargada con el contenido del registro *A*; la instrucción tiene la forma LD *(nn),A*.

En las dos instrucciones que acabamos de citar se puede utilizar el contenido del par *HL* en lugar de *nn*; las instrucciones se convierten en LD *A,(HL)* y LD *(HL),A*.

Todo *rr* puede ser cargado con cualquier *nn*; la instrucción tiene la forma LD *rr,nn*.

Todo *rr* puede ser cargado con el contenido de una posición de memoria y la siguiente; la instrucción tiene la forma LD *rr,(nn)*.

Una posición de memoria y la siguiente pueden ser cargadas con el contenido de un par de registros; la instrucción tiene la forma LD *(nn),rr*.

Usando el par *HL* se puede reducir la longitud de las dos instrucciones precedentes en un byte.

La llamada a una subrutina se efectúa con CALL *nn*. La llamada puede ser a cualquier dirección accesible de la memoria.

Toda subrutina debe terminar con un RET.

Las instrucciones CALL y RET utilizan la pila.

Se puede saltar a cualquier posición de la memoria mediante la instrucción JP *nn*.

En la instrucción precedente se puede dar la dirección del salto mediante el contenido de *HL*; se ocupa así 1 byte en lugar de 3. La instrucción toma entonces la forma JP *(HL)*.

La magnitud de un salto relativo se cuenta a partir del comienzo de la siguiente instrucción y debe estar en el intervalo de +127 a -128. La forma de la instrucción es JR *n*.

Los números de 16 bits se almacenan en memoria en orden invertido. El número está formado por 4 cifras hexadecimales; las dos más significativas se almacenan en la posición alta (posterior) de memoria y las dos menos significativas en la posición baja. En un par de registros el número se almacena en la forma natural (*high* o alto en *H* y *low* o bajo en *L* para el caso del par *HL*).

Los diversos códigos de estas operaciones y la función que realizan están en el apéndice A del libro.



## Aritmética elemental

En el capítulo precedente hemos visto las instrucciones LD r,n y LD r,r, que permitían cargar un número de 8 bits en un registro, o bien un registro en otro. Existe también un surtido completo de instrucciones para sumar y restar; la estructura de los códigos es semejante a la de las instrucciones LD r,n y LD r,r.

El registro A se denomina acumulador. Algunas instrucciones de carga de un registro sólo son posibles usando el acumulador. Pero donde este registro adquiere verdadera importancia es en las operaciones aritméticas de 8 bits, ya que es el único registro que almacena el resultado de estas operaciones.

Antes de estudiar las verdaderas operaciones aritméticas nos referiremos a dos instrucciones con un cierto contenido matemático; pueden ser ejecutadas con cualquier registro de uso general. La primera incrementa en 1 el contenido del registro; la segunda lo decrementa en 1. Sus códigos son INC r y DEC r, donde r es un registro de uso general. Por ejemplo, para un registro cuyo contenido sea 99, la instrucción INC lo transformará en 100 y la instrucción DEC en 98. También se puede hacer lo mismo con una dirección de la memoria, apuntando a ésta con el par HL; las instrucciones son entonces INC (HL) y DEC (HL). Los códigos de estas operaciones son los que se muestran en la figura 6.1.

Se reconocen en los bits 5, 4 y 3 de la codificación binaria los códigos de 3 bits correspondientes a los diferentes registros.

El funcionamiento de INC y DEC no presenta complicaciones, salvo una muy leve en dos casos. Si el contenido de un registro es 255, una instrucción INC lo convierte en 0. ¿Por qué? Ocurre como en los relojes: las horas aumentan da 1 a 23, pero la hora que sigue a 23 no es 24 sino 0. Observe que el contenido del registro es 1111 1111 en binario y que debe aumentar en 0000 0001; el resultado será 1 0000 0000, pero sólo se pueden cargar 8 bits; se adivina así la lógica de la operación en este caso.

El otro caso se da cuando un registro contiene el valor 0 y se efectúa con él la operación DEC. Usted mismo puede averiguar lo que ocurre entonces si tiene en cuenta que el registro está cargado con el valor 0000 0000 y que

ENSAMBLADOR	DECIMAL	HEX	BINARIO
INC B	4	04	00 000 100
INC C	12	0C	00 001 100
INC D	20	14	00 010 100
INC E	28	1C	00 011 100
INC H	36	24	00 100 100
INC L	44	2C	00 101 100
INC (HL)	52	34	00 110 100
INC A	60	3C	00 111 100

ENSAMBLADOR	DECIMAL	HEX	BINARIO
DEC B	5	05	00 000 101
DEC C	13	0D	00 001 101
DEC D	21	15	00 010 101
DEC E	29	1D	00 011 101
DEC H	37	25	00 100 101
DEC L	45	2D	00 101 101
DEC (HL)	53	35	00 110 101
DEC A	61	3D	00 111 101

de este valor se debe restar 1. Si no encuentra la solución, relea en el capítulo 3 la parte relativa a la notación de complemento a 2.

Para comprobar el funcionamiento de estas instrucciones puede escribir el programa de la figura 6.2, el cual, mediante las líneas 30 y 90, carga en memoria un programa en código de máquina y luego lo ejecuta en la línea 70. La línea 60 contiene una rutina en BASIC que produce el mismo efecto que la de código de máquina. La figura 6.3 contiene los códigos hexadecimal y nemotécnico de la rutina.

Notará que aparece antes de RET una instrucción que todavía no hemos explicado (lo haremos en el próximo capítulo); sin embargo, sabiendo que

```

10 MM=HIMEM
20 MEMORY 43799
30 FOR N=43800 TO 43811:READ D:POKE N,D:
A=A+D:NEXT
40 IF A<>1353 THEN CLS:PEN 3:PRINT "ERRO
R EN DATA":PEN 1:EDIT 90
50 INPUT "PULSE ENTER PARA EMPEZAR";A=A=
32:B=224
60 PRINT CHR$(A);:A=A+1:B=B-1:IF B<>0 TH
EN 60
70 PRINT:CALL 43800
90 DATA 6,224,62,32,205,90,187,60,5,32,2
49,201
100 END

```

Figura 6.2

la rutina de la línea 60 realiza las mismas operaciones, es posible que llegue a comprender su significado. Las letras NZ significan 'no cero' y se refieren al resultado de la última operación aritmética efectuada. La instrucción produce entonces un salto relativo a la etiqueta PRINT cuando el resultado de la última operación aritmética efectuada ha sido diferente de 0.

HEX	ENSAMBLADOR
06 E9	LD B,224
3E 20	LD A,32
CD 5A BB	PRINT: CALL 47962
3C	INC A
05 DEC	B
20 F9	JR NZ,,PRINT
C9	RET

Figura 6.3

La ejecución del programa producirá la impresión en la pantalla del juego de caracteres del Amstrad, comenzando por el espacio en blanco (el 32) y siguiendo con todos los caracteres del apéndice 3 de la Guía del usuario.

Pasaremos ahora a las instrucciones que sirven para sumar algo al registro

A o restar algo de él. En su forma más sencilla, estas instrucciones son muy simples. Para la suma se utiliza el código nemotécnico ADD y para la resta SUB. Como sólo se puede utilizar el registro A para la aritmética de 8 bits, parece innecesario especificar el registro; de hecho, esto es así para SUB, pero no para ADD, que también se puede utilizar para sumar 16 bits usando el par HL, como explicaremos más adelante. En la práctica no puede existir en ningún caso confusión respecto al sentido de ADD; por eso algunos ensambladores no requieren que se especifique el registro. Por el contrario, el ensamblador de DEVPAC no acepta ADD si no va seguido del registro.

Para sumar o restar del acumulador un número de 8 bits las instrucciones son ADD A,n y SUB n; los códigos completos son:

	ENSAMBIADOR	DECIMAL	HEX	BINARIO
ADD	A,n	198 n	C6 n	11 000 110 n
SUB	n	214 n	D6 n	11 010 110 n

Para probar estas instrucciones se pueden cambiar las líneas 30 y 90 del programa anterior por

```
30 FOR N=43800 TO 43812:READ D:POKE N,D:A=A+D:NEXT
90 DATA 6,224,62,32,205,90,187,198,1,5,32,248,201
```

sustituyendo 1353 por 1491 en la línea 40. Así se cambia la instrucción INC A por la equivalente ADD A,1. Como ahora hay un byte más, ha sido necesario aumentar en 1 la magnitud del salto relativo. Cuando se ejecute el programa el resultado será el mismo que antes; sin embargo, el programa ocupa un byte más.

Para probar la instrucción SUB los cambios son

```
50 INPUT "PULSE ENTER PARA EMPEZAR";A:A=255:B=224
60 PRINT CHR$(A);:A=A-1:B=B-1:IF B<>0 THEN 60
90 DATA 6,224,62,255,205,90,187,214,1,5,32,248,201
```

además de sustituir 1491 por 1529 en la línea 40. Los códigos hexadecimal y nemotécnico de la rutina que resulta son los que aparecen en la figura 6.4.

Aunque se trata de algo que estudiaremos sobre todo en el próximo capítulo, vamos a referirnos brevemente a la acción sobre los indicadores que tienen las últimas instrucciones que hemos visto. Cada *indicador* es un bit del *registro de estado* del microprocesador; el registro de estado se denota por F (de flag). Un indicador puede contener un 0, en cuyo caso se dice que 'está a 0' o puede contener un 1, y se dice que 'está a 1'. El *indicador de cero* (*zero*



```

06 E9          LD    B,224
3E FF          LD    A,255
CD 5A BB      PRINT: CALL 47962
D6 01          SUB   1
05             DEC   B
20 F8          JR    NZ,PRINT
C9             RET

```

Figura 6.4

*flag*) detecta si el resultado de la última operación realizada ha sido 0; este indicador suele ser representado por Z. Si el resultado de la operación ha sido 0, este indicador se pone 'a 1', o sea, activado; si el resultado de la operación ha sido diferente de 0, el indicador se pone 'a 0'. Se suele denotar estas dos alternativas por Z y NZ.

Otro indicador es el *indicador de arrastre (carry flag)*, que se suele representar por C. En las operaciones aritméticas de 8 bits sirve para detectar si la operación ha necesitado un noveno bits; en ese caso el indicador se pone 'a 1'. Si no es así, el indicador se pone 'a 0'. Se suele denotar estas dos alternativas por C y NC. Una suma de 8 bits activa el indicador de arrastre cuando el resultado es mayor que 255. Una resta lo hace cuando el resultado es menor que 0.

Las instrucciones INC y DEC modifican el indicador de cero en el sentido adecuado al resultado de la instrucción. Las instrucciones ADD y SUB modifican tanto el indicador de cero como el de arrastre.

De la misma manera que se puede sumar o restar al acumulador un número de 8 bits, también se puede sumar y restar el contenido de cualquier registro de uso general o del propio acumulador. Se tienen así las instrucciones ADD A,r y SUB r, cuyos códigos completos son:

ENSAMBLADOR	DECIMAL	HEX	BINARIO
ADD A, r	128 - 135	80 - 87	10 000 r
SUB r	144 - 151	90 - 97	10 010 r

La r representa el código de 3 bits correspondiente al registro; se trata de los códigos que ya vimos, es decir, de

B=000      C=001  
 D=010      E=011  
 H=100      L=101  
               A=111

También se puede utilizar en esta ocasión el código 110 para representar el contenido de la dirección de memoria a la que apunta el par HL. Esto proporciona las operaciones ADD A,(HL) y SUB (HL).

El funcionamiento de SUB r y ADD A,r es similar al de ADD A,n y SUB n, incluso en la forma en que afectan las instrucciones a los indicadores de cero y de arrastre.

Vamos a proponerle un ejercicio sencillo que usted podrá realizar con los conocimientos adquiridos hasta ahora. Le proponemos que escriba una rutina que sume el contenido de la dirección de memoria 43894 con el de la dirección 43896 y coloque el resultado en la dirección 43898; convendrá además que la rutina esté diseñada para ser cargada a partir de la posición 43850. Cuando haya terminado la rutina, deberá cargarla en la memoria. Utilice un ensamblador, si dispone de él. Si no, codifique la rutina utilizando la información que hemos dado hasta ahora y cárguela desde un programa BASIC similar al que hemos venido utilizando.

Si no ha conseguido escribir la rutina le proporcionaremos algunas soluciones posibles en este capítulo. Si ha podido realizarla, deberá comprobar que funciona correctamente. Vamos a ver en qué forma puede hacerse esta comprobación, que exigirá lógicamente imprimir los resultados en la pantalla. En primer lugar termine su rutina con:

ENSAMBLADOR	DECIMAL	HEX
CALL 43800	205 24 171	CD 18 AB
RET	201	C9

Con ello llama a una subrutina que aún no existe; es la que deberá introducir a continuación y que le proporcionamos en la figura 6.5. Esta rutina sirve para imprimir el resultado.

Para introducir esta rutina puede emplear un programa BASIC como el que ya hemos utilizado; debe entonces tener en cuenta que la rutina consta de 35 bytes y que la suma de comprobación es 3966. También debe recordar que la ejecución debe comenzar en 43850.

En cualquier caso, por si usted carece de ensamblador, le proporcionaremos en el apéndice B de este libro un programa, que hemos llamado CAR-GADOR HEX, que le servirá para introducir todas las rutinas que le daremos en este libro. Dedique el tiempo necesario a cargarlo y grabarlo en cinta. Aprenderá en seguida a utilizarlo, ya que la mecánica es siempre la misma:

	ENSAMBLADOR	DECIMAL	HEX
	ORG 43800		HIMEM EN AB17
	ENT 43800		DIR INIC AB18
	LD A,(43898)	58 122 171	3A 7A AB CHECK
	LD L,A	111	6F
	LD H,0	38 0	26 00
	LD DE,-100	17 156 255	11 9C FF
	CALL REDN	205 44 171	CD 046D
			2C AB
	LD E,-10	30 246	1E F6
	CALL REDN	205 44 171	CD 2C AB
	LD A,L	125	7D
	JR PRIN	24 9	18 09 042D
REDN:	LD A,0	62 0	3E 00
FNUM:	INC A	60	3C
	ADD HL,DE	25	19
	JR C,FNUM	56 252	38 FC
	SBC HL,DE	237 82	ED 52
	DEC A	61	3D
PRIN:	ADD A,#30	198 48	C6 0409
			30
	CALL 47962	205 90 187	CD 5A BB
	RET	201	C9 END 02DB

Figura 6.5

el cargador le solicitará la dirección en que comienza la parte protegida de la memoria, la dirección en que comienza la rutina, y luego, sucesivamente, los bytes de la rutina en codificación hexadecimal. Cuando se teclea END en lugar de un byte, termina la introducción de la rutina. Cada 10 bytes le pedirá la suma de los mismos para su comprobación. Cuando en el libro le proporcionemos una rutina, le daremos también las sumas de comprobación. En la rutina de la figura 6.5 estas sumas eran 046D, 042D, 0409 y 02DB.

Después de cargar la rutina, el problema consistirá en introducir en la memoria los datos que se deben sumar. Para ello recurriremos a BASIC. Copie el programa

```

400 INPUT "PRIMER NUMERO";A:INPUT "SEGUNDO NUMERO";B
410 PRINT A;"+";B;"=";
420 POKE 43894,A:POKE 43896,B
430 CALL 43800
440 GOTO 400

```

Figura 6.6

y ejecútelo con GOTO 400, El programa le pedirá los datos, los cargará en la memoria y llamará a su rutina; el resultado se imprimirá en la pantalla. Cuando desee terminar el programa, pulse la tecla [ESC] dos veces.

Habrás observado que, cuando se suman dos números cuya suma sobrepasa 255, la respuesta es incorrecta. Recordará que ya hemos explicado por qué, y también que entonces se activa el indicador de arrastre. La forma en que esto tiene solución se comprende mejor cuando se reflexiona sobre la manera en que habitualmente sumamos números. Si se desea hacer la suma  $9+6+8$ , lo que se hace es

$9+6=15$  con 1 de arrastre

$5+8=13$  con 1 de arrastre,

luego la respuesta es 3 con 2 de arrastre, o sea, 23, Lo mismo ocurre con una suma binaria: se suma por columnas y cuando se arrastre un 1 se lo añade a la siguiente columna.

```

      1010 0101 (165)
+     1011 0000 (176)
      1
      +0
      = 1

      01
      +0
      =01

      101
      +0
      = 101

      0101
      +0
      =0101

      0 0101
      +1
      = 1 0101

      11 0101
      +1
      = 101 0101

      101 0101
      +0
      +0
      = 101 0101

      1101 0101
      +1
= 0101 0101 (85) = con 1 de arrastre (256), o sea, 341

```

Al terminar la suma hay un arrastre de una unidad: su valor relativo es de 256 veces el valor del bit menos significativo.

Así pues, lo que se requiere para sumar números más grandes es una serie de bytes: el arrastre de cada byte se debe añadir entonces al byte siguiente.

Hay instrucciones que permiten hacer esto automáticamente. Se trata de las instrucciones de 'suma con arrastre' y 'resta con arrastre', cuyos códigos nemotécnicos son ADC y SBC (C de *carry*). Cuando se realizan estas operaciones se incluye automáticamente en la suma o resta el valor del indicador de arrastre.

Pensemos, por ejemplo, en el programa de la figura 6.7

```
LD HL,43896
LD A,(43894)
ADD A,(HL)
LD (43898),A
```

```
LD A,(43895)
INC HL
ADD A,(HL)
LD (43899),A
```

Figura 6.7

imaginando que 43894 tiene 1010 0101 (165), 43896 tiene 1011 0000 (176) y que las restantes direcciones tienen 0. La primera parte del programa sumará 165 con 176 y almacenará en 43898 el resultado, que es 85. La segunda parte sumará los contenidos de 43895 y 43897 (o sea, 0 + 0), almacenando el resultado en 43899. Veamos qué ocurre ahora si cambiamos la segunda instrucción ADD por ADC. La primera parte será la igual, pero en la segunda la suma será 0+0+arrastre y el resultado, que es 1, se almacenará en 43899. De esta manera se obtiene la suma correcta en las direcciones 43898 y 43899. Si se añade la instrucción LD HL,(43898) al final del programa, el registro L cargará el byte menos significativo y el registro H el más significativo; de esta manera el par HL contendrá el valor 0000 0001 0101 0101b o 01 55 Hex que corresponde a la suma correcta.

Las instrucciones de suma y resta con arrastre tienen los códigos

ENSAMBLADOR	DECIMAL		HEX		BINARIO
ADC A,n	206	n	CE	n	11 001 110 n
SBC A,n	222	n	DE	n	11 011 110 n
ADC A,r	136 - 143	88 - 8F	10	001	r
SBC A,r	152 - 159	98 - 9F	10	011	r

ASSEMBLER	HEX
ORG 43850	HIMEM EN AB17
ENT 43850	DIR INIC AB4A
LD HL, 43896	SUMA
CALL 47896	21 78 AB
ADD A, (HL)	CD 18 BB
LD (43898), A	86
LD A, (43895)	32 7A AB 04C1
INC HL	3A 77 AB
ADC A, (HL)	23
LD (43899), A	8E
CALL 43800	32 7B AB
	CD 18 044A
	AB
RET	C9 END 0174
	MAS? S/N S
ORG 43800	DIR INIC AB18
	SUMA
LD HL, (43898)	2A 7A AB
NOP	00
NOP	00
NOP	00
NOP	00
NOP	00
NOP	00
LD DE, -1000	11 0160
	18 FC
CALL REDN	CD 36 AB
LD DE, -100	11 9C FF
CALL REDN	CD 36 0571
	AB
LD E, -10	11 F6 FF
CALL REDN	CD 36 AB
LD A, L	7D
JR PRIN	18 09 04FD
REDN: LD A, 0	3E
	00
FNUM: INC A	3C
ADD HL, DE	19
JR C, FNUM	38 FC
SBC HL, DE	ED 52
DEC A	3D
PRIN: ADD A, #30	C6 0409
CALL 47962	30 CD 5A BB
RET	
	C9 END 02DB
	MAS? S/N N

Figura 6.8

Como de costumbre, r representa cualquier registro de uso general, A o (HL); los códigos son también los de siempre. Observe que SBC requiere que se precise el registro A, lo que no ocurría con SUB; esto se debe a que el código SBC admite otras interpretaciones que veremos más adelante.

Para comprobar el funcionamiento de las instrucciones ADC y SBC introduzca el programa de la figura 6.8.

Este programa es demasiado largo para utilizar el método del DATA, así que hemos omitido el código decimal. Utilice el CARGADOR HEX a falta de un ensamblador. Para ejecutar el programa utilice el comando R del ensamblador o bien la instrucción CALL 43850 desde el sistema operativo.

Lo que hace el programa es sumar el código ASCII de la primera tecla que se pulse con el contenido de la dirección 43896, almacenando el resultado en la posición 43898. A continuación se suma 'con arrastre' el contenido de 43895 y 43897 y el resultado se almacena en 43899. La lectura de la tecla pulsada se realiza mediante la llamada CALL 47896 a una rutina del sistema operativo que espera que se pulse una tecla y almacena su código ASCII en el acumulador.

Si no ha colocado nada en las direcciones cuyo contenido se suma, lo único que obtendrá como respuesta será el código ASCII de la tecla que pulse, como podrá comprobar consultando la tabla del apéndice 3 de la guía del usuario.

Para realizar otras pruebas deberá cargar algo en dichas posiciones, sitúese en BASIC, utilizando el comando B si está utilizando el ensamblador. Entonces, para sumar 220 y 89 por ejemplo, pulse

```
POKE 43896,220[ENTER] CALL 43850 [ENTER]
```

y luego SHIFT y Y (el código de Y es 89); obtendrá 309 como respuesta. Para sumar 23260 y 345 pulse

```
POKE 43896,220[ENTER] POKE 43897,90[ENTER]
POKE 43895,1[ENTER] CALL 43850[ENTER]
```

y luego SHIFT y Y. La respuesta debería ser 23605, pero no lo es; ¿por qué? En realidad todo ha sido hecho correctamente. 23260 es 5ADCh y ha sido correctamente introducido: el byte bajo DCh, que es 220, en 43896; el byte alto 5Ah, que es 90, en 43897. El 345 es 0159h; se ha introducido 59h, que es 89, como código de la Y; el 1 se ha introducido en 43895. El programa ha sumado el código de la Y con el contenido de 43896, colocando el resultado en 43898. Como  $59h + DCh = 135h$ , en 43898 debe haber 35h o 53. Eso es lo que ocurre, como se puede comprobar con

```
?PEEK(43898) [ENTER]
```

Además, el indicador de arrastre estará a 1. A continuación el programa ha sumado los contenidos de 43895 y 43897 y el bit de arrastre. Como  $5Ah + 01h + \text{arrastre} = 5Ch$  o 92, el programa habrá colocado en 43899 el número 92. Se puede comprobar que esto ha ocurrido así. La respuesta de la suma es entonces  $92 * 256 + 53 = 23552 + 53 = 23605$  que es lo correcto, pero no lo que ha aparecido en la pantalla.

La respuesta hay que buscarla en la segunda parte del programa, que es la parte que se encarga de visualizar el resultado. De hecho, tantos bytes con la instrucción NOP (que no hace nada) le habrán sugerido posiblemente que la verdadera intención es rellenar este espacio más adelante.

Hay dos instrucciones (ADD HL,DE y SBC HL,DE) que no hemos descrito todavía. Daremos una idea de ellas, aunque vamos a explicarlas con más detalle en los siguientes capítulos.

La instrucción SUB se utiliza exclusivamente con números de 8 bits, pero las instrucciones ADD, ADC y SBC se pueden utilizar con números de 16 bits. Para ello, el acumulador A se sustituye por el par HL, que desempeña así el papel de acumulador; contiene uno de los números que se operan y almacena después el resultado de la operación. El segundo de los números que se operan debe estar en alguno de los pares BC, DE o HL, o también en el registro de 16 bits SP (el puntero de pila). Sin embargo este segundo número no puede ser dado explícitamente, ni indicado como contenido en una dirección de memoria, ni siquiera dando esta dirección mediante un par de registros; en otras palabras, no existen instrucciones del tipo ADD HL,23456, ADC HL,(23456) o SBC HL,(DE). La lista de las operaciones posibles, con sus códigos, es la que se muestra en la figura 6.9.

Funcionan como sus equivalentes ADD A,B ADC A,B y SBC A,B, salvo por el hecho de que trabajan con números de 16 bits. Por ejemplo, para sumar los números 55536 y 2000 se puede utilizar la siguiente sucesión de instrucciones:

```
LD DE,55536
LD HL,2000
ADD HL,DE
```

Tras la ejecución de estas instrucciones, el par HL contendrá la suma 57536 (E0C0h en H C0h en L) y el par DE contendrá el sumando 55536 (D8F0h D8h en D F0h en E); el indicador de arrastre quedará a 0. Si la suma realizada hubiera sido  $55536 + 23605$ , la respuesta correcta 79141 (13525h) no habría podido ser almacenada en 16 bits; por lo tanto la respuesta habría sido 13605 (3525h) y el indicador de arrastre habría quedado a 1. El valor del bit de arrastre sería en ese caso de 65536 (o sea,  $2^{16}$ ) veces el del bit menos significativo de par de registros, mientras que para las operaciones de 8 bits este valor es de 256 (o sea,  $2^8$ ) veces el del bit menos significativo.



ENSAMBLADOR	DECIMAL	HEX	BINARIO
ADD HL,BC	9	09	00 001 001
ADD HL,DE	25	19	00 011 001
ADD HL,HL	41	29	00 101 001
ADD HL,SP	57	39	00 111 001
ADC HL,BC	237 74	ED 4A	11 101 101 01 001 010
ADC HL, DE	237 90	ED 5A	11 101 101 01 011 010
ADC HL, HL	237 106	ED 6A	11 101 101 01 101 010
ADC HL, SP	237 122	ED 7A	11 101 101 01 111 010
SBC HL,BC	237 66	ED 42	11 101 101 01 000 010
SBC HL,DE	237 82	ED 52	11 101 101 01 010 010
SBC HL,HL	237 98	ED 62	11 101 101 01 100 010
SBC HL,SP	237 114	ED 72	11 101 101 01 110 010

Figura 6.9

Cuando se desea realizar la suma o resta incluyendo el bit de arrastre, se deben utilizar las instrucciones ADC o SBC. Sin embargo, no existe la instrucción SUR para 16 bits. En consecuencia, si se desea efectuar una resta sin restar al mismo tiempo el bit de arrastre, hay que poner a 0 el indicador de arrastre si se encuentra activado.

Hay instrucciones que permiten activar (poner a 1) el indicador de arrastre, y también para poner este indicador a su valor complementario (el contrario del que tenga); son las instrucciones SCF y CCF. Por el contrario, no existe una instrucción para poner a 0 el indicador de arrastre. Lo que se puede hacer es ponerlo a 1 y complementarlo después, o sea, efectuar SCF y CCF. Lo que se hace habitualmente es utilizar para esta finalidad la instrucción lógica AND A, que es más breve; explicaremos esta instrucción en el capítulo 8.

Volvamos ahora a nuestro programa de suma; como ya vimos, realizaba correctamente la operación pero imprimía un resultado incorrecto. La primera parte del programa almacenaba el byte alto de la respuesta en la dirección 43899, y el byte bajo en 43898.

La segunda parte comienza por LD HL,(43898), que carga en L el contenido de la posición indicada, y en H el de la posición siguiente. La instrucción cargará 35h en L y 5Ch en H, haciendo HL=5C35h o 23605, que es lo correcto. El problema no reside en esta instrucción. Tampoco está en las 6 instrucciones NOP, que no tienen ningún efecto.

A continuación se carga DE con -1000, que es FC18h (de momento no se verá a qué conduce esto); luego se produce una llamada a la rutina que comienza en la etiqueta REDN. Vamos a examinar en detalle las operaciones que se producen.

- 1) LD A,0 hace  $A=0$ .
- 2) INC A hace  $A=A+1$
- 3) ADD HL,DE realiza la suma de  $HL=5C35h$  y  $DE=FC18h$ . 5C35h es 23605. FC18h es 64536 en decimal o -1000 si se interpreta en complemento a 2.  $23605+64536=88141$ . Como el mayor número que cabe en 16 bits es 65535, el indicador de arrastre se pone a 1. Además,  $88141-65536=22605$ , luego el efecto final será restar 1000 del contenido de HL.
- 4) JR C,FNUM produce el salto a la etiqueta FNUM si el indicador de arrastre está a 1. En tal caso el efecto que se produce es incrementar el contenido de A en 1 y volver a restar 1000 del contenido de HL. El registro A contará el número de veces que se ha repetido esta operación.
- 5) SBC HL,DE Se llega a esta instrucción cuando ya no existe arrastre en ADD HL,DE. En ese caso se devuelve a HL el número 1000 que se había restado (restar un número negativo equivale a sumar).
- 6) DEC A anula el último incremento de A. El resultado ahora es que en A está el número de veces que HL contenía a 1000, y en HL el resto de la división por 1000. En nuestro caso estos valores son  $HL=605$  y  $A=23$ .
- 7) ADD A,#30 suma a A el número hexadecimal 30 (para el ensamblador de Highsoft el símbolo # significa hexadecimal). En nuestro caso 23 (17h) más 30h (48) da 71 (47h).
- 8) CALL 47962 llama a la rutina de la ROM que se encarga de escribir el carácter cuyo código figura en A.
- 9) RET señala el fin de la rutina.

Lo que se pretende es escribir la primera cifra decimal del resultado, o sea, el número de miles que hay en HL. Como las cifras de 0 y 9 tienen por códigos ASCII los que van de 30h a 39h, todo hubiese marchado bien si este número de miles hubiera estado entre 0 y 9. Pero como era 23, el resultado ha sido escribir la letra G, cuyo código es 71, en lugar de las cifras 2 y 3. Vamos a ver cómo se puede arreglar el programa.

Si está usando el ensamblador, escriba CALL 30004 [ENTER] y a continuación L[ENTER] para listar el programa. Introduzca las dos nuevas instrucciones que le damos más abajo en el lugar de las dos primeras NOP y borre las cuatro restantes NOP. A continuación escriba A [ENTER] [ENTER] [ENTER] para ensamblar de nuevo el programa.

Si no dispone de ensamblador, reemplace los seis bytes con el CARGADOR HEX, suministrándole AB17 como valor para HIMEM y AB1B como dirección inicial. Cargue así las instrucciones

ENSAMBLADOR		HEX	
LD	DE, -10000	11	F0 D8
CALL	REDN	CD	36 AB END 0387
		MAS?	S//N N

Ejecute ahora el programa y verá como trabaja perfectamente con números cuya suma quepa en 16 bits (hasta 65535).

Puede cambiar la primera parte del programa para experimentar con las restantes instrucciones de suma y resta de 8 bits. Mientras siga utilizando (HL) para señalar las direcciones que almacenan los números, y *no* utilice instrucciones que usen explícitamente n o nn, le bastará con cambiar el byte que contiene la instrucción. Recuerde que en código de máquina no ocurre como en BASIC, en el que se pueden insertar instrucciones.

Si ha entendido bien todo esto, no le resultará difícil escribir programas para sumar o restar dos números cualesquiera utilizando operaciones de 8 bits. Otra cosa será conseguir visualizar el resultado. Si desea una impresión en pantalla le bastará con modificar el programa que hemos utilizado.

En este tipo de tareas es donde se observan las ventajas del trabajo con 16 bits. Las sumas de números de 16 bits proporcionan resultados que ocupan 17 bits; mientras que el resultado de un producto necesita 32 bits. La ventaja está en que trabajar con 16 bits para obtener resultados de 32 bits no requiere más instrucciones que para obtener 24 bits con aritmética de 16 y 8 bits.

Observe que con 32 bits se pueden representar números hasta  $4294967295$  ( $2^{32}$ ).

Puede parecer molesta la imposibilidad de utilizar operandos numéricos en las instrucciones aritméticas de 16 bits, pero esto es fácil de solucionar. De hecho, la primera parte del programa de la figura 6.8, que utilizaba aritmética de 8 bits, se puede escribir también como muestra la figura 6.10.

Esta alternativa utiliza 21 bytes, uno menos que la original. Además, conserva el resultado en HL, lo que ahorra posteriormente la instrucción LD HL,(43898), de 3 bytes, a la hora de ejecutar la rutina de impresión.

<b>ORE</b>	<b>43850</b>	HIMEM	EN	AB17
<b>ENT</b>	<b>43850</b>	DIR INIC	AB18	
<b>LD</b>	<b>HL, (43895)</b>	21 77	AB SUMA	
<b>LD</b>	<b>A, (HL)</b>	7E		
<b>INC</b>	<b>HL</b>	23		
<b>LD</b>	<b>E, (HL)</b>	5E		
<b>INC</b>	<b>HL</b>	23		
<b>LD</b>	<b>D, (HL)</b>	56		
<b>LD</b>	<b>H, A</b>	67		
<b>CALL</b>	<b>47896</b>	CD		03EF
			18	BB
<b>LD</b>	<b>L, A</b>	6F		
<b>ADD</b>	<b>HL, DE</b>	19		
<b>LD</b>	<b>(4389B), HL</b>	22	7A	AB
<b>CALL</b>	<b>43800</b>	CD	18	AB 0432
<b>RET</b>		C9		END 00C9

Figura 6.10

También se puede mejorar utilizando instrucciones de carga de 16 bits, como hacemos en el programa de la figura 6.11. Así se emplean solamente 19 bytes.

El programa que hemos escrito puede ser un buen ejercicio, pero no es una rutina útil. Para que lo fuese, debería ser una rutina utilizable por un programa en circunstancias cualesquiera, lo que no es el caso por estar ligada a posiciones concretas de memoria en las que deben figurar los números que se suman. Lo que podríamos hacer es reescribir la rutina de manera que se limite a sumar los números que haya en los pares HL y DE, a escribir el resultado en pantalla y a conservarlo en el par HL. De esta manera, si por ejemplo estamos realizando un juego de marcianos en el que se pueden obtener puntuaciones de 10, 20, 50, 100 y 400, lo que haríamos cada vez que se elimina un invasor es cargar su valor en DE y llamar a la rutina; el total de puntua-

<b>ORG</b>	<b>4 3 8 5 0</b>	HIMEM	EN	AB17
<b>ENT</b>	<b>4 3 8 5 0</b>	DIR	INIC	AB18
<b>LD</b>	<b>H L , ( 4 3 8 9 6 )</b>	21 78 AB	SUMA	
<b>LD</b>	<b>A , ( 4 3 8 9 5 )</b>	3A 77	AB	
<b>LD</b>	<b>D , A</b>	57		
<b>CALL</b>	<b>4 7 8 9 6</b>	CD 18	BB	0496
<b>LD</b>	<b>E , A</b>	5F		
<b>ADD</b>	<b>H L , D E</b>	19		
<b>LD</b>	<b>( 4 3 8 9 8 ) , H L</b>	22 7A	AB	
<b>CALL</b>	<b>4 3 8 0 0</b>	CD 18	AB	
<b>RET</b>		C9	END	0418

Figura 6.11

<b>ORG</b>	<b>43830</b>	HIMEM	EN	AB17
<b>ENT</b>	<b>43850</b>	DIR	INIC	AB18
<b>LD</b>	<b>HL, (43896)</b>	21 78 AB	SUMA	
<b>LD</b>	<b>A, (43995)</b>	3A 77	AB	
<b>LD</b>	<b>D, A</b>	57		
<b>CALL</b>	<b>47896</b>	CD 18	BB	0496
<b>LD</b>	<b>E, A</b>	5F		
<b>AND</b>	<b>A</b>	A7		
<b>SBC</b>	<b>HL, DE</b>	ED	52	
<b>LD</b>	<b>(43898), HL</b>	22 7A	AB	
<b>CALL</b>	<b>43800</b>	CD 18	AB	051C
<b>RET</b>		C9	EMD	00C9

Figura 6.12

ción acumulada se iría así escribiendo en la pantalla y quedaría acumulado en HL para una nueva suma.

Las rutinas de las figuras 6.10 y 6.11 pueden servir también para restar números, pero hay que tener en ese caso la precaución de poner a 0 el indicador de arrastre para no falsear inadvertidamente los resultados. Para ello se puede utilizar la instrucción AND A, como ya dijimos. Es lo que hacemos en el ejemplo de la figura 6.12, que es como el de 6.11 pero modificado para la resta; resta el contenido de DE de HL y deja el resultado en HL.

Como ejercicio final de este capítulo, el lector puede escribir un programa que sume números de 16 bits y almacene el resultado en la memoria como un número de 32 bits. Si es usted capaz de escribir este programa, nosotros le ayudaremos a comprobar su funcionamiento si respeta algunas premisas: almacene el resultado en las posiciones de memoria que van de la 43896 a la 43899, de menos significativo a más significativo; haga que el programa comience en 43840 (AB40h) y que termine con CALL 43700 y RET.

La figura 6.13 contiene la rutina que llamará su programa para comprobar que funciona correctamente; está tal y como lo muestra el listado del ensamblador, para evitar errores. Si utiliza el ensamblador debe cargar la columna de códigos nemotécnicos, cuidando de añadir el símbolo ':' detrás de las etiquetas.

```

Hisoft  GENA3 Assembler. Page      1.

Pass 1 errors: 00

10 ; SUBROUTINA PARA IMPRIMIR EN
    DECIMAL UN NUMERO DE 32 BITS
20
AAB4 30 ORG 43700
AAB4 40 ENT 43700
AAB4 2A7BAB 50 LD HL,(43896)
AAB7 223EAB 60 LD (43838),HL
AABA 2A7AAB 70 LD HL,(4389B)
AABD 2240AB B0 LD (43840),HL
AAC0 110036 90 LD DE,#3600; BYTES BAJOS
AAC3 0165C4 100 LD BC,#C465;
    BYTES ALTOS DE -1 000 000 000
AAC6 CD0CAB 110 CALL REDN
AAC9 11001F 120 LD DE,#1F00; BYTES BAJOS Y
AACC 010AFA 130 LD BC,#FA0A;
    BYTES ALTOS DE 0 00 000 000
AACF CD0CAB 140 CALL REDN
AAD2 118069 150 LD DE,#6980; BYTES BAJOS Y
AAD5 0167FF 160 LD BC,#FF67;
    BYTES ALTOS DE -10 000 000
AAD8 CD0CAB 170 CALL REDN
AADB 11C0BD 180 LD DE,#BDC0; BYTES BAJOS Y
AADE 01F0FF 190 LD BC,#FFF0;

```

AAE1	CD0CAB	200		BYTES ALTOS DE - 1 000 000
AAE4	116079	210	CALL	REDN
			LD	DE, #7960 ;
				BYTES BAJOS Y
AAE7	11FEFF	220	LD	BC, #FFFE ;
				BYTES ALTOS DE - 100 000
AAEA	CD0CAB	230	CALL	REDN
AAED	11F0DB	240	LD	DE, -10000 ; BYTES BAJO
AAFO	01FFFF	250	LD	BC, #FFFF ;
				BYTES ALTOS DE
AAF3	CD0CAB	260	CALL	REDN
AAF6	1118FC	270	LD	DE, -1000
AAF9	CD0CAB	280	CALL	REDN
AAFC	119CFF	290	LD	DE, -100
AAFF	CD0CAB	300	CALL	REDN
AB02	1EF6	310	LD	E, -10
AB04	CD0CA8	320	CALL	REDN
AB07	3A3EAB	330	LD	A, (43838)
AB0A	1B25	340	JR	PRIN
AB0C	3E00	350	LD	A, 0
AB0E	3C	360	INC	A
AB0F	2A3EAB	370	LD	HL, (43B38)
AB12	19	380	ADD	HL, DE
AB13	223EAB	390	LD	(43838), HL
AB16	2A40AB	400	LD	HL, (43B40)
AB19	ED4A	410	ADC	HL, BC
AB1B	2240AB	420	LD	(43B40), HL
AB1E	3BEE	430	JR	C, FNUM
AB20	2A3EAB	440	LD	HL, (43838)
AB23	ED52	450	SBC	HL, DE
AB25	223EAB	460	LD	(43838), HL
AB28	2A40AB	470	LD	HL, (43840)
AB2B	ED42	480	SBC	HL, BC
AB2D	2240AB	490	LD	(43B40), HL
AB30	3D	500	DEC	A
AB31	C630	510	ADD	A, «30
AB33	CD5ABB	520	CALL	47962
AB36	C9	530	RET	

## Resumen

Vamos a resumir las instrucciones explicadas en este capítulo. Utilizaremos los símbolos:

- r = cualquiera de los registros de 8 bits (A, B, C, D, E, H o L)
- rr = cualquier par de registros que se utilicen como uno de 16 bits
- n = un número de 8 bits, o sea, entre 0 y 255
- nn = un número de 16 bits, o sea, entre 0 y 65535
- ( ) rodeando un número o un par de registros = el contenido de la dirección.
- PC = contador de programa
- SP = puntero de pila

INC r y DEC r suman 1 o restan 1 a r; el resultado afecta al indicador de cero. Si el resultado es 0, el indicador se pone a 1; si no, a 0.

INC rr y DEC rr hacen lo mismo, pero con un par de registros; estas instrucciones no afectan a los indicadores.

El acumulador A es el único registro que sirve para almacenar el resultado de las operaciones aritméticas de 8 bits.

Las operaciones aritméticas de 8 bits son:

SUBr SUBn SUB (nn) SUB (HL) para restar una cantidad de A.  
ADD A,r ADD A,n ADD A,(nn) ADD A,(HL) para sumar a A una cantidad.

SBC A,r SBC A,n SBC A,(nn) SBC A,(HL) para restar con arrastre una cantidad de A.

ADC A,r ADC A,n ADC A,(nn) ADC A,(HL) para sumar con arrastre una cantidad a A.

Se debe utilizar el par HL para almacenar el resultado de las operaciones aritméticas de 16 bits.

Las operaciones aritméticas de 16 bits son:

ADD HL,rr para sumar al par HL el contenido del par rr.

ADC HL,rr para sumar con arrastre al par HL el contenido del par rr.

SBC HL,rr para restar con arrastre el contenido de rr del par HL.

Todas estas operaciones aritméticas afectan al indicador de arrastre según sea el resultado de la operación. Lo mismo ocurre con el indicador de cero, salvo para la instrucción ADD de 16 bits, que no le afecta.

Si no se desea que la instrucción SBC se efectúe con el bit de arrastre, se puede poner a 0 el indicador de arrastre mediante la instrucción AND A.



## 7

### Indicadores, condiciones y decisiones condicionadas

Ya hemos visto algo del funcionamiento de los indicadores de cero y de arrastre (Z y C) en relación con las operaciones aritméticas. Cada uno de estos indicadores es un bit del *registro de estado (flag)*, que se denota por F. Puesto que este registro es de 8 bits, se puede sospechar que existirán otros indicadores; así es. La estructura del registro de estado F es la siguiente:

•   SIGNO •   M/A	• Z CERO • Z/NZ	NO SE USA	H SEMI- ARRAS TRE.	NO SE USA	P/V PARIDAD/ SOBREPASAM IENTO PE/PO	• SUMA/ RESTA •	C ARRASTRE C/N C
----------------------------	-----------------------------	--------------	-----------------------------	--------------	---	--------------------------	------------------------

La letra que hay sobre cada indicador es la abreviatura usada por el fabricante del microprocesador, Zilog, para representarlo; es el símbolo con el que se identifica cada indicador en las tablas del apéndice A. Después viene el nombre del indicador; los nombres que se utilizan en inglés son:

SIGNO	es	SIGN
CERO	es	ZERO
SEMIARRASTRE	es	HALF CARRY
PARIDAD/SOBREPASAMIENTO	es	PARITY/OVERFLOW
SUMA/RESTA	es	ADD/SUBTRACT
ARRASTRE	es	CARRY

Para cuatro de los indicadores, figuran también los símbolos con que se representan los estados posibles del indicador. Sólo estos cuatro indicadores son accesibles al usuario; los restantes los utiliza internamente el Z80.

En las situaciones más diversas, existen decisiones cuyo signo depende de que se den o no determinadas condiciones. Son las denominadas *decisiones condicionadas*. En código de máquina, el recurso de que dispone el programa para saber si se dan o no ciertas condiciones son los indicadores. Dado que sólo hay 4 indicadores accesibles, se requiere algo de ingenio para comprobar con ellos un amplio abanico de condiciones.

Supongamos por ejemplo que al programar un juego necesitamos comparar el tanteo obtenido con el tanteo más alto hasta el momento, para la realizar la sustitución si se ha batido el récord. Lo que podemos hacer es restar de la nueva puntuación el antiguo récord y observar el indicador de arrastre. Si no se activa (o sea, si se produce NC), sabremos que se ha obtenido un nuevo récord. Pero hay un problema: si el récord anterior era 15575 y el nuevo 21024, sabremos así que se ha obtenido un nuevo récord, pero habremos perdido ambas cifras para quedarnos con 5449, que es la diferencia. Hay formas de solucionar este inconveniente.

Lo ideal sería realizar una falsa resta, es decir, una instrucción que active los indicadores como si fuese una resta pero sin realizar la operación. Existe una instrucción de este tipo y se llama *comparación*. Su código nemo-técnico es CP y funciona como la instrucción SUB salvo por el hecho de que no altera el valor de los registros, excepción hecha del registro de estado. La instrucción SUB podía realizarse solamente con el registro A: lo mismo ocurre con CP.

El comportamiento de los indicadores tras la instrucción CP es también el mismo que en caso de SUB.

Su código se construye como el de las operaciones de 8 bits; ahora bien, los bits 5, 4 y 3 llevan 111 en el caso de CP. Así se tiene

SUB n	10 010 110 n	8	000
CP n	10 111 110 n	C	001
		D	010
SUB r	10 010 r	E	011
CP r	10 111 r	H	100
		L	101
SUB (HL)	10 010 110	(HL)	110
CP (HL)	10 111 110	A	111

Los indicadores se utilizan para la toma de decisiones, esto es, para ejecutar alternativamente unas u otras instrucciones en función de una condición impuesta acerca del estado de un indicador. Lo análogo en BASIC es la instrucción 'IF condición THEN instrucción a ejecutar'. La analogía va aún más allá; lo que suele ponerse tras THEN es una instrucción GOTO de salto (aunque la palabra GOTO puede generalmente omitirse, como ocurre en el Amstrad). Lo mismo ocurre en código de máquina. El programa de la figura

6.3 realizaba un salto (JR) dependiendo de una condición sobre el indicador de cero. El programa de la figura 6.5 hacía lo mismo, pero con el indicador de arrastre.

Puede parecer que son pocas las comprobaciones que se pueden realizar, pero no es así. Veremos que los indicadores sirven para 'indicar' muchas cosas.

Examinaremos primero el indicador de arrastre, sobre el que ya tenemos cierta experiencia.

Salvo INC y DEC, todas las operaciones que provoquen el sobrepasamiento del registro o registros sobre los que actúan, hacen que se active el indicador de arrastre. Por el contrario, este indicador se desactiva cuando la operación no ha producido este sobrepasamiento. Por ejemplo, LD A,0 y DEC A no modificarían el indicador de arrastre, mientras que LD A,0 y SUB 1, en este orden, sí lo activarían. Las secuencias

LD B,156	LD BC,65000	LD BC,65000
LD A,100	LD HL,5536	LD HL,5536
ADD A,B	ADC HL,BC	SBC HL,BC

activan el indicador de arrastre, mientras que las secuencias

LD BC,5536	LD A,225
LD HL,65000	y ADD A,25
SBC HL,BC	

desactivan dicho indicador. En el caso de la instrucción CP, ésta activará el indicador de arrastre cuando el número n, o el contenido de r o de la posición HL sean superiores al contenido del acumulador A y lo desactivará en caso contrario. El indicador de arrastre se emplea en código de máquina con una finalidad similar a la de los operadores > y < de BASIC.

Todas las operaciones aritméticas afectan al indicador de cero, salvo la ADD de 16 bits. Este indicador se activa cuando el resultado de la operación es 0 y se desactiva en caso contrario. La instrucción CP activa el indicador de cero cuando las cantidades que se comparan son iguales y lo desactiva en caso contrario. Por eso el indicador de cero se emplea en código de máquina de la misma manera que el operador-de BASIC.

Además de las instrucciones INC y DEC de 8 bits, hay otras instrucciones que afectan al indicador de cero sin modificar el indicador de arrastre; ya iremos viendo estas instrucciones. En lo sucesivo, al presentar una instrucción nueva, diremos en qué forma afecta a los indicadores accesibles al programador.

Mediante una programación adecuada, es posible contestar a todas las cuestiones relativas al programa que se respondan con sí o no, comprobando

los indicadores de cero y de arrastre. En ocasiones bastará con la comprobación de un sólo indicador; otras requerirán varias comprobaciones complementarias.

Claro que esto no es fácil de hacer al principio, e incluso proporcionará en un primer momento resultados diferentes de los previstos; pero es posible lograrlo si se aprende a pensar un poco como lo hace el microprocesador.

Observe el ejemplo siguiente. Su finalidad es averiguar si el valor almacenado en el acumulador A corresponde a algún código ASCII, si es el código de una letra y si es el código de la letra 'A'; según sea el caso, el programa saltará a las etiquetas:

```

NOTASC si no se trata de un código ASCII;
NOTLET si no es el código de ninguna letra;
ISA si es el código de la 'A';
ISLET si es el código de una letra diferente.

```

Las 'preguntas' se van realizando en el siguiente orden: 1) ¿contiene A un código ASCII?; 2) si lo contiene, ¿es el código de una letra?; 3) si es así, ¿se trata del código de la primera letra del alfabeto? La secuencia de instrucciones es la siguiente:

CP 128	Los códigos ASCII válidos van de 0 a 127.
JR NC,NOTASC	Si el registro A almacena un valor que no es un código ASCII (128 o superior), se pondrá a 0 (simbólicamente NC) el indicador de arrastre; el programa saltará entonces a la etiqueta NOTASC. Si A almacena un código ASCII, existirá arrastre y el programa pasará a la instrucción siguiente.
CP 32	Los códigos ASCII para letras son todos superiores a 31.
JR C,NOTLET	Si el valor almacenado en A es igual o menor que 31, se habrá activado el indicador de arrastre (simbólicamente C); el programa saltará a NOTLET.
CP 65	Se compara con el código de la letra 'A'.
JR Z,ISA	Si se produce la igualdad, se activa el indicador de cero (simbólicamente Z) y el programa saltará a ISA.

Cabría pensar que si el programa no ha realizado el salto en ningún momento, lo que hay en el acumulador es el código de una letra diferente de 'A' y que por lo tanto el programa debe saltar a ISLET, pero esto no es así. Hay códigos entre 32 y 127 que no corresponden a letras. De hecho, sólo son letras los códigos 65...90 y 97...122.

Cambiando CP 32 por CP 65 y eliminando la CP 65 de dónde está, se mejora un poco la situación. Pero quedan aún las lagunas 91...96 y 123...127 por evitar. Esto se consigue añadiendo ahora las siguientes instrucciones:

CP 123

JR NC,NOTLET Si el valor de A es igual o mayor que 123 (luego está en 123... 127), se trata de un código que no es de una letra.

CP 91

JR C,ISLET Si el valor de A es menor que 91 (luego está en 66...90), se trata de una letra diferente de 'A'.

CP 97

JR C,NOTLET Si el valor de A es menor de 97 (luego está en 91...96), se trata de un código que no es de una letra.

Normalmente, si se trata de distinguir una 'A' pulsada en el teclado, conviene aceptar 'a' tanto como 'A'. Para que así sea, hay que añadir una instrucción al programa.

JR Z,ISA Si el valor de A es exactamente 97, el código corresponde a la 'a'.

Si el programa ha superado todos los saltos, el contenido de A estará en el intervalo 98... 122 y será una letra minúscula diferente de 'a'. Por lo tanto, la etiqueta ISLET se debe colocar justamente en este punto, evitando así un nuevo salto.

Introduzca este programa y experimente con él. Cuando lo entienda perfectamente, cambie de letra. Naturalmente, el ensamblador le permitirá realizar fácilmente las modificaciones precisas, mientras que con el CARGADOR HEX deberá volver a cargar todo el programa.

El programa que presentamos en la figura 7.1 no es más que el que acabamos de comentar, pero completado para permitir la entrada del valor mediante el teclado e imprimir ciertos mensajes elegidos según sea la entrada efectuada. Observe la manera de escribir mensajes y de elegirlos; analizaremos esto más adelante.

Si utiliza el CARGADOR HEX ya sabrá que el valor para HIMEM debe ser AAB3 y la dirección inicial AAB4; los códigos hexadecimales que se introducen son los de la segunda columna de la figura, que comienza por CD18BB. Las sumas de comprobación que pedirá el programa son 05EA, 0380, 036C, 023A, 0567, 0395, 02DB, 0226, 02A5, 0248, 0264, 01C8.

Si utiliza el ensamblador no es necesario que divida los mensajes en trozos pequeños; nosotros lo hemos hecho así porque el ensamblador sólo lista en

AAB4		30	ORG	43700
AAB4		20	ENT	43700
AAB4	CD1BBB	30	START	CALL
AAB7	0604	40	LD	B,4
AAB9	FEFC	50	CP	252
AABB	C8	60	RET	Z
AABC	FE80	70	CP	128
AABE	3016	80	JR	NC,NOTASC
AACO	FE41	90	CP	65
AAC2	3811	100	JR	C,NOTLET
AAC4	2811	110	JR	Z,ISA
AAC6	FE7B	120	CP	123
AAC8	300B	130	JR	NC,NOTLET
AACA	FE5B	140	CP	91
AACC	3806	150	JR	C,ISLET
AACE	FE61	160	CP	97
AADO	3803	170	JR	C,NOTLET
AAD2	2803	180	JR	Z,ISA
AAD4	05	190	ISLET	DEC
AAD5	05	200	NOTLET	DEC
AAD6	05	210	NOTASC	DEC
AAD7	21EDAA	220	ISA	LD
AADA	3E0A	230	LD	A,#0A
AADC	BE	240	LOOKMS	CP
AADD	23	250	INC	HL
AAD6	20FC	260	JR	NZ,LOOKMS
AAEO	10FA	270	DJNZ	LOOKMS
AAE2	7E	280	PRINT	LD
AAE3	CD5AB8	290	CALL	47962
AAE6	FE0A	300	CP	#0A
AAE8	2BCA	310	JR	Z,START
AAEA	23	320	INC	HL
AAEB	18F5	330	JR	PRINT
AAED	0A	340	MESST	DEFB
AAEE	41204C45	350	DEFM	"A LE "
AAF2	54544552	360	DEFM	"TTER"
AAF6	20425554	370	DEFM	" BUT"
AAFA	204E4F54	380	DEFM	" NOT"
AAFE	2041	390	DEFM	" A"
AB00	ODOA	400	DEFW	#0A0D
AB02	4E4F5420	410	DEFM	"NOT "
AB06	41204C45	420	DEFM	"A LE "
AB0A	54544552	430	DEFM	"TTER"
AB0E	ODOA	440	DEFM	#0A0D
AB10	4E4F5420	450	DEFM	"NOT "
AB14	41534349	460	DEFM	"ASCII"
AB18	49	470	DEFM	"I "
AB19	ODOA	480	DEFW	#0A0D
AB1B	594F5520	490	DEFM	"YOU "
AB1F	50524553	500	DEFM	"PRES"
AB23	53454420	510	DEFM	"SED "
AB27	4121	520	DEFM	"A ! "
AB29	ODOA	530	DEFW	#0A0D

Figura 7.1

hexadecimal los cuatro primeros bytes de cada línea. La línea 350 podría haber sido perfectamente

DEFM "A LETTER BUT NOT A"

suprimiendo entonces las líneas 360 a 390.

Los mensajes que genera el programa son los siguientes:

"A LETTER BUT NOT A" (una letra diferente de A)

"NOT A LETTER" (no es una letra)

NOT ASCII" (no es un código ASCII)

"YOU PRESSED A!" (ha pulsado la A)

Corresponden a las posibles alternativas que analizaba el programa.

Algunos puntos especiales del programa merecen un comentario.

Cuando un programa en código de máquina realiza un bucle sin fin, su ejecución no se detiene salvo que se apague el ordenador. Para que no ocurra esto, conviene preparar una salida del programa. En nuestro caso esta salida se produce cuando, tras la ejecución de la rutina WAIT KEY de la dirección 47896, el acumulador queda cargado con el valor 252, que es el código que genera la tecla [ESC]. En ese caso se ejecuta una instrucción RET.

La sección siguiente del programa es la que ya ha sido comentada; termina por enviar el programa a una de las cuatro etiquetas que hemos descrito. Pero conviene hacer notar que el registro B ha sido cargado con el número 4, y que el resultado de saltar a una u otra etiqueta es hacer que B llegue a la línea 220 con un valor entre 1 y 4 que dependerá de la etiqueta. Este es el comienzo del mecanismo que permite elegir el mensaje apropiado.

El par HL se coloca entonces en la dirección de la etiqueta MESST, que es el comienzo de los mensajes. El byte 0Ah marca la separación entre uno y otro mensaje. El mecanismo selector consiste entonces en disminuir B en una unidad cada vez que se encuentra el byte 0Ah, hasta que el valor de B se haga 0, en cuyo caso comienza a escribirse el mensaje. Aunque no se observe en esta zona ninguna instrucción que disminuya B en una unidad, lo que ocurre es que la instrucción está implícita en una nueva instrucción que no habíamos mencionado todavía: la instrucción DJNZ.

La instrucción DJNZ actúa como las instrucciones DEC B y JR NZ juntas, pero ocupa un byte menos que ellas y además no afecta a los indicadores. Sus códigos son

ENSAMBLADOR	DECIMAL	HEX	BINARIO			
DJNZ n	16 n	10 n	00	001	010	n

El número *n* representa, como en los saltos relativos, la magnitud del salto contada desde el comienzo de la instrucción siguiente.

Obsérvese que cada marca 0Ah de fin de mensaje está precedida del byte 0DH, que es el código que, al ser impreso, produce un salto de línea que dispone el cursor en la posición adecuada para el próximo mensaje.

Finalmente, el control vuelve al comienzo del programa y el proceso se repite para 3a siguiente tecla pulsada.

Antes de comenzar la explicación de otro nuevo indicador, vamos a dar una relación completa de las instrucciones condicionales de salto relativo. Para este tipo de salto sólo se pueden utilizar condiciones relativas a los indicadores de cero y de arrastre. El detalle de estas instrucciones y de sus códigos es el siguiente:

ENSAMBLADOR		DECIMAL	HEX	BINARIO
DJNZ	<i>n</i>	16 <i>n</i>	10 <i>n</i>	00 010 000 <i>n</i>
JR	<i>n</i>	24 <i>n</i>	18 <i>n</i>	00 011 000 <i>n</i>
JR	NZ, <i>n</i>	32 <i>n</i>	20 <i>n</i>	00 100 000 <i>n</i>
JR	Z, <i>n</i>	40 <i>n</i>	28 <i>n</i>	00 101 000 <i>n</i>
JR	NC, <i>n</i>	48 <i>n</i>	30 <i>n</i>	00 110 000 <i>n</i>
JR	C, <i>n</i>	56 <i>n</i>	3B <i>n</i>	00 111 000 <i>n</i>

Figura 7.2

Como es fácil de imaginar, también los saltos absolutos, JP, pueden convertirse en saltos condicionados al valor de los indicadores. Lo mismo ocurre con las instrucciones CALL y RET, pero con éstas se pueden utilizar condiciones referidas a los cuatro indicadores accesibles al programador.

Ya hemos explicado cómo funcionan los indicadores de cero y de arrastre; veamos cómo lo hacen los dos restantes.

El *indicador de signo (sign flag)* se representa por S; sus dos alternativas son la de *signo negativo (minus sign)* que pone a 1 el indicador y se representa por M, y la de *signo positivo (plus sign)* que pone a 0 el indicador y se representa por P.

El *indicador de paridad/sobrepasamiento (parity/overflow flag)* se representa por P/V; sus dos alternativas son la de *paridad par (parity even)* que pone a 1 el indicador y se representa por PE, y la de *paridad impar (parity odd)* que pone a 0 el indicador y se representa por PO.



Como usted recordará, los registros de uso general tenían asociado un código de 3 bits, que se utilizaba para formar los códigos binarios de las instrucciones. Lo mismo ocurre con las condiciones sobre los indicadores. Estos códigos son:

NZ	no cero (not zero)	000
Z	cero (zero)	001
NC	sin arrastre (no carry)	010
C	arrastre (carry)	011
PO	paridad impar (parity odd)	100
PE	paridad par (parity even)	101
P	signo positivo (plus sign)	110
M	signo negativo (minus sign)	111

En el cuadro que sigue, las letras cc representan una de estas condiciones; en el código binario, cc se debe sustituir por su código de 3 bits.

ENSAMBLADOR		BINARIO			
JP	cc, nn	11	cc	010	nn
CALL	cc, nn	11	cc	100	nn
RET	cc	11	cc	000	

Así, por ejemplo,

JP NC,47962 es 11 010 010 0101 1010 1011 1011  
y CALL Z,47960 es 11 001 100 0101 1000 1011 1011

Lo que indica el indicador de signo es, obviamente, el signo del resultado de una operación. Ahora bien, sólo tiene este significado cuando el resultado deba interpretarse escrito en la notación de complemento a 2. Para lo que se utiliza en cualquier caso este indicador es para comprobar el valor del bit 7 de un byte. Por ejemplo, si el registro A contiene el número 254 después de una operación aritmética, el indicador de signo reflejará signo negativo puesto que el bit 7 de A es 1; sin embargo, puede ser erróneo interpretar esto en el sentido de que el resultado es un número negativo. En lo que sigue, emplearemos a veces la expresión 'entero con signo' para referirnos a un entero que hay que interpretar en notación de complemento a 2.

Todas las instrucciones aritméticas de 8 bits, incluyendo CP (la comparación), las INC y DEC de 8 bits y las instrucciones ADC y SBC de 16 bits afectan al indicador de signo. No le afecta ninguna de las restantes instrucciones que hemos visto hasta ahora. Para las nuevas instrucciones que vaya-

mos introduciendo se indicará en qué medida afectan a los indicadores, en particular al de signo. El apéndice A describe también la influencia sobre los indicadores de todas las instrucciones.

El indicador de paridad/sobrepasamiento tiene, como indica su nombre, un doble significado. De hecho, lo que tiene es uno de los dos significados dependiendo de la instrucción (pero no ambos al mismo tiempo).

Todas las instrucciones que afectan al indicador de cero afectan al indicador de paridad/sobrepasamiento, y todas las que hemos visto por ahora lo hacen en el sentido de indicador de sobrepasamiento.

El indicador de sobrepasamiento se activa cuando en un cálculo, interpretado como cálculo de un número con signo, el resultado sobrepasa el tamaño en que debe ser almacenado; se desactiva cuando esto no ocurre. El concepto es un poco complicado y vamos a explicarlo con algún ejemplo. El programa

```
LD    A,-80
ADD  A,-80
```

tiene por efecto almacenar en A el número binario 0110 0000, que es % 0 60h y no es el resultado esperado, ya que es un número positivo. Nótese que en este caso se habrá activado el indicador de arrastre. El programa

```
LD    A,80
ADD  A,80
```

almacenaría en A el número 1010 0000, que es -96, y desactivaría el indicador de arrastre. En ambos programas queda activado el indicador de sobrepasamiento.

Así pues, el indicador de sobrepasamiento señala el exceso en las operaciones con signo (en complemento a 2) mientras que el de arrastre señala el exceso en las operaciones de números positivos. Como dejan claro los ejemplos anteriores, estos dos indicadores son completamente independientes uno de otro.

Las operaciones aritméticas que producen resultados fuera del intervalo  $-128 \leq n \leq 127$  en el caso de 8 bits, y de  $-32768 \leq n \leq 32767$  en el de 16 bits, activan el indicador de sobrepasamiento. Obsérvese que dos números de signo diferente no pueden originar sobrepasamiento cuando se suman. Por el contrario, dos números del mismo signo no pueden dar sobrepasamiento cuando se restan.

En los códigos nemotécnicos, los símbolos que se emplean son PE para sobrepasamiento y PO para no sobrepasamiento. No son símbolos nada nemotécnicos (ni siquiera en inglés) pero es que se usan los mismos que para la paridad.

Cuando se emplea este indicador como indicador de paridad (no hemos visto aún operaciones que lo afecten en este sentido) lo que mide es la parí-

dad del número de bits iguales a 1 en un byte. El indicador se activa (PE) cuando hay un número par de bits 1 en el byte y se desactiva cuando dicho número es impar.

Veamos por fin dos instrucciones de las que ya hemos hablado, SCF y CCF. La instrucción SCF (*set carry flag*) tiene por efecto poner a 1 el indicador de arrastre. La instrucción CCF (*complement carryflag*) cambia el valor del indicador de arrastre a su valor contrario (cualquiera que fuera el valor anterior del indicador). Los códigos de estas instrucciones son:

ENSAMBLADOR	DECIMAL	HEX	BINARIO
CCF	63	3F	00 111 111
SCF	55	37	00 110 111

## Resumen

Vamos a resumir las instrucciones explicadas en este capítulo. Utilizaremos los símbolos:

- r =cualquiera de los registros de 8 bits (A, B, C, D, E, H o L)
- rr =cualquier par de registros que se utilicen como uno de 16 bits
- n =un número de 8 bits, o sea, entre 0 y 65535
- nn =un número de 16 bits, o sea, entre 0 y 65535
- () rodeando un número o un par de registros=el contenido de la dirección.
- PC= contador de programa
- SP =puntero de pila

Los indicadores accesibles al programador son C (arrastre), Z (cero), S (signo) y P/V (paridad/sobrepasamiento).

El indicador de sobrepasamiento señala el hecho de que en una operación aritmética de números con signo, el resultado ha cambiado de signo y es incorrecto.

cc puede ser C, NC, Z, NZ, PE, PO, M y P.

CP realiza una falsa resta (SUB) del registro A y altera los indicadores en consecuencia, pero no cambia ninguna otra cosa.

JR sólo admite condiciones sobre los indicadores C y Z.

DJNZ equivale a DEC B y JR NZ, pero no altera los indicadores.

JP, CALL y RET pueden convertirse en condicionadas al valor de un indicador.

Ninguna de las instrucciones LD, CALL, JP, JR o RET afecta a los indicadores.



## Operaciones lógicas

El microprocesador Z80 posee un juego de instrucciones lógicas similar al de operadores lógicos del BASIC del Amstrad, lo que no es sorprendente ya que es justamente el Z80 el que realiza el trabajo cuando se está ejecutando un programa BASIC. Como usted estará familiarizado con la utilización de los AND, OR y XOR de BASIC, nos resultará más sencillo explicar sus análogos de código de máquina. Si no es así, le convendría leer lo que sobre este aspecto se dice en el capítulo 4 de la Guía del usuario, así como practicar un poco. En lo que sigue supondremos que se conocen bien las expresiones lógicas del BASIC del Amstrad.

Las instrucciones lógicas AND, OR y XOR se consideran instrucciones aritméticas; sólo pueden ser utilizadas para valores de 8 bits y usando el registro A. Los códigos son semejantes a los de las restantes operaciones aritméticas de 8 bits; los bits 5, 4 y 3 son los que determinan la naturaleza de la operación. El código nemotécnico no requiere que se haga referencia al registro A, ya que en este aspecto no puede haber confusión, como ocurriría con SUB. Las instrucciones lógicas afectan a los indicadores en el sentido que corresponda al resultado de la operación. El de arrastre queda siempre a 0, ya que AND, OR y XOR no pueden producir un resultado que precise más de 8 bits. La consideración de sobrepasamiento en estas instrucciones carece de sentido, de manera que el indicador P/V se interpreta como indicador de paridad. El indicador de signo refleja el estado del bit 7 de A tras la operación. El indicador de cero se activa cuando A no tiene ningún bit a 1 y se desactiva en caso contrario.

ENSAMBLADOR	DECIMAL	HEX	BINARIO
AND n	230	E6	11 100 110
AND r	160 - 167	A0 - A7	10 100 r
XOR n	238	EE	11 101 110
XOR r	168 - 175	AS - AF	10 101 r
OR n	246	F6	11 110 110
OR r	176 - 183	B0 - B7	10 110 r

Para entender bien la utilidad de las operaciones lógicas hay que empezar por pensar en binario; sólo así se comprende el sentido de muchos de los aspectos en los que se las puede utilizar. Por ello es muy probable que usted no alcance a ver ahora toda la utilidad que tienen estas instrucciones.

Volvamos al programa de la figura 7.1. La comprobación de los códigos se hacía independientemente para las letras mayúsculas, para las minúsculas y para el intervalo entre ambas. Pero de hecho, la única diferencia entre los dos tipos de letras está en el bit 5 de su código. Para las mayúsculas es un 0 y para las minúsculas un 1. Con la instrucción AND es posible convertir todas las letras en mayúsculas, y con OR se pueden convertir todas las letras en minúsculas. ¿De qué manera? Podrá verlo a través de las modificaciones que vamos a realizar en el programa de la figura 7.1.

Cambie la línea 220 del programa por

	HEX	ENSAMBLADOR
AAD7	CD 2B AB	CALL EXTRA

que requiere 01A3 como suma de comprobación, y añadida al final del programa

AB2B	CD 5A BB	EXTRA	CALL 47962
AB2E	00		NOP
AB2F	00		NOP
AB30	CD 5A BB		CALL 47962
AB33	3E 20		LD A,32 ; THE CODE FOR SPACE
AB35	CD 5A BB		CALL 47962
AB38	21 ED AA		LD HL,MESST
AB3B	C9		RET

Esta última parte tiene las sumas 0422, 0463.

Al ejecutar ahora el programa, el caracter correspondiente a la tecla pulsada aparecerá repetido dos veces, seguido de un espacio y del correspondiente mensaje. Las dos instrucciones NOP le proporcionan espacio para que pueda experimentar con AND, OR y XOR y vea el efecto que producen. Comience por cambiar las dos NOP por

	HEX	ENSAMBLADOR
AB2E	F6 20	OR #20

Si no dispone de ensamblador, lo más sencillo será que utilice POKE &AB2E,&F6:POKE&AB2F,&20 como comando directo.

Ejecute el programa probando con varias teclas y pulsando unas veces sí y otras no la tecla [SHIFT]. (Asegúrese de que no está activada [CAPS LOCK] ya que Amstrad no ha puesto un indicador luminoso o que nos permita saberlo). Verá ahora que las mayúsculas cambian a minúsculas, las minúsculas y los números quedan como están y los símbolos cambian o no según sea el bit 5 de su código. Incorporando la instrucción OR#20 al programa principal se ahorran unas cuantas instrucciones CP.

La versión reformada del programa de la figura 7.1 está en la figura 8.1. Ahora se utiliza el indicador de signo para saber cuándo no se trata de un código ASCII (si el bit 7 vale 1 el código será 128 o superior)- La instrucción OR sirve indirectamente para activar (si es el caso) el indicador de signo, sin necesidad de un CP 0 que habría añadido un byte al programa. Ahora ha habido un ahorro de un byte tras sustituir JR por JP.

Hisoft GENA3 Assembler. Page 1.

Pass 1 errors: 00

```

1 ; FIG 8,1
2 * OTRA VERSION DEL PROGRAMA DE 7.1

AAB4      10      ORE      43700
AAB4      20      ENT      43700
AAB4      30      START   CALL 47896
AAB7      40      LD       B,4
AAB9      50      CP       252
AABB      60      RET      Z
AABC      90      OR       #20
AABE      100     JP       M,NOTASC
AAC1      120     CP       123
AAC3      130     JR       NC,NOTLET
AAC5      160     CP       97
AAC7      170     JR       C,NOTLET
AAC9      180     JR       Z,ISA
AACB      190     ISLET    DEC  B
AACC      200     NOTLET   DEC  B
AACD      210     NOTASE   DEC  B
AACE      220     ISA     LD   HL,MESST
AAD1      230     LD       A,#0A
AAD3      240     LOOKMS   CP   (HL)
AAD4      250     INC      HL
AAD5      260     JR       NZ,LOOKMS
AAD7      270     DJNZ     LOOKMS
AAD9      280     PRINT    LD   A,(HL)
AADA      290     CALL    47962
AADD      300     CP       #0A
AADF      310     JR       Z,START
AAE1      320     INC      HL
AAE2      330     JR       PRINT

```

AAE4	OA	340	MESST	DEFB	#OA
AAE5	41204C45	350		DEFM	"A LE "
AAE9	54544552	360		DEFM	"TTER"
AAED	20425554	370		DEFM	" BUT"
AAF1	204E4F54	3B0		DEFM	" NOT"
AAF5	2041	390		DEFM	" A"
AAF7	ODOA	400		DEFW	#0A0D
AAF9	4E4F5420	410		DEFM	"NOT "
AAFD	41204C45	420		DEFM	"A LE "
AB01	54544552	430		DEFM	"TTER"
AB05	ODOA	440		DEFW	#0A0D
AB07	4E4F5420	450		DEFM	"NOT "
AB0B	41534349	460		DEFM	"ASCII"
ABOF	49	470		DEFM	"i "
AB10	ODOA	4B0		DEFW	#0A0D
AB12	594F5520	490		DEFM	"YOU "
AB16	50524553	500		DEFM	"PRES"
AB1A	53454420	510		DEFM	"SED "
AB1E	4121	520		DEFM	" A ! "
AB20	ODOA	530		DEFW	#0A0D

Pass 2 errors: 00

Table used: 110 from 184

Executes: 43700

**Figura 8.1.** Sumas de comprobación: 0582, 05B8, 0215, 04B6, 0439, 02A7, 022B, 02A2, 0251, 0268, 020D, 0608, 0278.

En lugar de la instrucción OR se puede usar AND para cambiar minúsculas en mayúsculas. La forma exacta de la instrucción para cambiar a 0 el bit 5 es AND #DF.

También se puede utilizar XOR en lugar de OR. En este caso las mayúsculas pasan a minúsculas y viceversa. Pero ahora hay que tener cuidado para no pulsar teclas que no sean alfanuméricas, ya que los códigos de algunas teclas se transforman con XOR en códigos de control.

La instrucción AND se puede utilizar para 'enmascarar' ciertos bits. Ésta es la terminología que se emplea cuando se ignoran determinados bits, convirtiéndolos en ceros. Por ejemplo, si en un programa se necesita que las letras lleven códigos del 1 (para A) al 26 (para Z), la solución es enmascarar los 3 bits superiores del código de la letra con AND%00011111.

La instrucción OR tiene el efecto opuesto y puede servir para recuperar los bits enmascarados por la instrucción AND. Una de las aplicaciones más frecuentes y apropiadas de esta instrucción es la 'sobreescripción' en pantalla; consiste en escribir sobre lo que ya está escrito sin suprimirlo. También se la utiliza, como ya hemos dicho, para recuperar bits enmascarados o modificados.



Por ejemplo, para pasar del valor de una cifra decimal a su código ASCII se puede utilizar la instrucción `ADDA,#30` y, de hecho, es lo que hicimos en el programa que fuimos desarrollando a lo largo del capítulo 6. Pero el mismo efecto se consigue con la instrucción `OR#30`, que es la que hubiésemos utilizado si la hubiésemos conocido entonces.

La instrucción `XOR` sirve para cambiar el valor de ciertos bits a su valor opuesto. Al igual que la `OR`, se la usa a menudo en rutinas de escritura en pantalla. Por ejemplo, el Amstrad la utiliza para la escritura 'transparente' (consulte el capítulo 5 de la Guía del usuario).

La siguiente instrucción lógica es la de complementación, `CPL`, cuya análoga en BASIC es el operador `NOT`. Sólo puede operar sobre el registro A. Su efecto es cambiar el valor de todos los bits al valor opuesto, o sea, tiene el mismo efecto que `XOR#FF`. La instrucción `CPL` no afecta a ninguno de los indicadores accesibles al programador.

El programa de la figura 8.2 realiza una demostración gráfica de la aplicación de `CPL`. Lo que hace es complementar todas las posiciones del 'mapa de pantalla', o sea, del área de la memoria en que se almacena la información que aparece en la pantalla. Se invierten los bits correspondientes a las tintas de papel y de pluma, lo que en modo 2 tiene el efecto de crear el negativo de la pantalla; sin embargo, en los modos 0 y 1 el efecto es más complejo, ya que admiten más de 2 colores de tinta. En modo 2, donde sólo hay 2 colores, cada byte controla 8 puntos de la pantalla (*pixels*), de manera que si el bit de un punto está a 0 su color es el de la tinta 0, y si está a 1 su color es el de la tinta 1. Al invertir los bits con `CPL`, lo que se hace justamente es invertir el número de la tinta que corresponde a cada punto.

En modo 1 hay 4 colores de tinta. La tinta que colorea cada punto de la pantalla se determina con 2 bits, de acuerdo con el código natural que asigna 00 para la tinta 0, 01 para la 1, 10 para la 2 y 11 para la 3. Cada byte controla entonces 4 puntos de la pantalla. Si, por ejemplo, la tinta del papel es la 0 y la de la pluma es la 1, después de la ejecución del programa el papel se verá del color de la tinta 3 y la pluma del de la tinta 2.

En modo 0 la cosa se complica más, puesto que hay 16 tintas a distinguir; cada punto necesita 4 bits y cada byte controla entonces 2 puntos.

Ahora se ve claramente por qué la resolución se hace más baja cuando aumenta el número de tintas que se emplean simultáneamente. Lo que ocurre además es que sólo en modo 2 los bits de un byte se corresponden con los puntos de la pantalla en el orden que pudiera esperarse. En los otros modos existe una mezcla de bytes que hace las cosas más complicadas. Por ejemplo, en modo 1, los bits 3 y 7 controlan el punto más a la izquierda de los que corresponden al byte, los bits 2 y 6 el que le sigue a la derecha, y así sucesivamente.

Hisoft GENA3 Assembler. Page 1.

Pass 1 errors: 00

```

1 ; FIG 8,2
2 ; PROGRAMA PARA COMPLEMENTAR
  EL MAPA DE PANTALLA

AAB4      10      ORG  43700
AAB4      20      ENT  43700
AAB4      2100CO  30      LD  HL,#C000
AAB7      7C      40      LD  A,H
AABB      B5      50      DR  L
AAB9      C8      60      RET  Z
AABA      7E      70      LD  A,(HL)
AABB      2F      80      CPL
AABC      77      90      LD  (HL),A
AABD      23      100     INC  HL
AABE      18F7    110     JR   LOOP

```

Pass 2 errors: 00

Sumas de comprobación:042J, 010F.

Figura 8.2

Hisoft GENA3 Assembler. Page

Pass 1 errors: 00

```

10 ; PROGRAMA PARA DIVIDIR LA PANTALLA
20 ; EN COLUMNAS DE COLORES
   INK  0,1,2,5

AAB4      30      ORG  43700
AAB4      40      ENT  43700
AAB4      2100CO  50      LD  HL,#C000
AAB7      7C      60      LD  A,H
AABB      B5      70      OR  L
AAB9      C8      80      RET  Z
AABA      3E5C    90      LD  A,%01011100
AABC      77      100     LD  (HL) , A
AABD      23      110     INC  HL
AABE      1BF7    120     JR   LOOP

```

Pass 2 errors: 00

Sumas de comprobación: 040E, 010F.

Figura 8.3

Para terminar de rizar el rizo, el orden en que se controla la pantalla no es el que uno pudiera esperar (salvo si se tiene la mente algo retorcida).

El programa de la figura 8.3 da otro ejemplo de manejo de la pantalla. Su efecto es dividir la pantalla del modo 1 en columnas cuya anchura es de un punto, coloreadas alternativamente de las cuatro tintas posibles. Aclaremos que los códigos de las cuatro tintas se almacenan con el bit más significativo del código en la posición menos significativa de las dos que corresponden al punto. Desde luego, quien diseñó esta pantalla debía tener algo de sádico.

El apéndice F explica detenidamente lo que se refiere al mapa de la pantalla.

La última de las instrucciones lógicas es la instrucción NEG (negación). El efecto que tiene es cambiar de signo el contenido del registro A, tomando el complemento a 2. En otras palabras, transforma A en la diferencia 0-A. Esta instrucción afecta a los indicadores como si se tratase de una instrucción SUB de 8 bits. Quedan afectados los indicadores C, Z, S y P/V, este último en el sentido de sobrepasamiento.

Los códigos de CPL y NEG son:

ENSAMBLADOR	DECIMAL	HEX	BINARIO
NEG	237 68	ED 44	11 101 101 01 000 100
CPL	47	2F	00 101 111

## Resumen

Vamos a resumir las instrucciones explicadas en este capítulo. Utilizaremos los símbolos:

- r = cualquiera de los registros de 8 bits (A, B, C, D, E, H o L)
- rr = cualquier par de registros que se utilicen como uno de 16 bits
- n - un número de 8 bits, o sea, entre 0 y 255
- NN = un número de 16 bits, o sea, entre 0 y 65535
- ( ) rodeando un número o un par de registros = el contenido de la dirección.
- PC = contador de programa
- SP = puntero de pila

Todas las instrucciones lógicas trabajan con el valor que haya en A. AND, OR y XOR se pueden utilizar con r o con n.

Con AND se ponen a 1 los bits que estaban a 1 a la vez en el acumulador y en el operando; los demás se ponen a 0.

Con OR se ponen a 1 los bits que estaban a 1 en el acumulador o en el operando; los demás se ponen a 0.

Con XOR se ponen a 1 los bits que estaban a 1 en el acumulador o en el operando; pero no en ambos; los demás se ponen a 0.

AND, OR y XOR ponen a 0 el indicador de arrastre y afectan a los restantes de acuerdo con el resultado que quede en el registro A. El indicador P/V tiene el sentido de indicador de paridad.

CLP y NEG no llevan operandos.

CLP cambia cada bit de A a su valor contrario. No afecta a los indicadores.

NEG devuelve el complemento a 2 del valor de A. Los indicadores quedan afectados como si se tratase de una instrucción SUB que restase 0-A.

## Utilización de la pila

Ya hemos introducido brevemente en el capítulo 5 el funcionamiento de la *pila (stack)*, motivados por la necesidad de comprender el funcionamiento de las instrucciones CALL y RET. La instrucción CALL deposita en la pila la dirección de la instrucción siguiente (que es previsiblemente la dirección de la vuelta); la instrucción RET recupera de la pila dicha dirección. Advertimos asimismo sobre la necesidad de cuidar el equilibrio entre la información que se almacena en la pila y la que sale de ella.

Existen instrucciones que permiten utilizar la pila como un almacén temporal de datos para el usuario. Se trata de una utilización compartida, ya que, simultáneamente, el programa continua almacenando en ella sus direcciones de retorno de las subrutinas. Por ello es necesario manejar con cuidado este tipo de instrucciones. Bien es verdad que en ocasiones se provoca deliberadamente el que una instrucción RET devuelva el programa a un punto diferente del de partida. Pero cuando de forma inadvertida se obtiene este resultado, es casi seguro que se provoque un fracaso irreparable del programa, cuya consecuencia inmediata será tener que apagar y volver a encender el ordenador. Viene al caso ahora recomendarle que grabe previamente el programa antes de ejecutarlo. Así, en caso de catástrofe, podrá al menos recuperar el programa para corregirlo.

Las instrucciones que permiten guardar datos en la pila y recuperarlos son, respectivamente, PUSH y POP. La instrucción PUSH *rr* coloca en la pila el contenido del par de registros *rr* y disminuye en dos unidades el puntero de pila SP para que siga apuntando al extremo de la pila. Por el contrario, la instrucción POP *rr* almacena en el par *rr* el contenido del extremo de la pila y aumenta en dos unidades el puntero de pila. La mecánica es la misma que ya estudiamos en la figura 5.8, pero el trasvase no se realiza al contador del programa, PC, sino a un par de registros.

Los códigos de estas instrucciones son

ENSAMBLADOR				BINARIO			
PUSH	<i>rr</i>	11		<i>rr</i> 0	101		
POP	<i>rr</i>		11	<i>rr</i> 0	001		

donde hay que sustituir rr por un par de registros y por el código binario de dicho par. Estos códigos binarios eran

BC = 00 DE=01 HL=10

Pero ahora se puede utilizar también el código 11b, que indica el par AF formado por el acumulador A y el registro de estado F.

Los códigos de PUSH y POP tienen gran semejanza (no casual) con CALL y RET:

CALL 11 001 101 RET 11 001 001  
PUSH 11 rr0 101 POP 11 rr0 001

En la figura 9.1 presentamos un programa que sirve para conocer la dirección a la que apunta el puntero de pila y el dato situado en el extremo de la pila (el que se obtendría haciendo una extracción de la pila).

Hisoft GENA3 Assembler. Page 1.

Pass 1 errors: 00

```

1 ; FIG 9,1
2 i PROGRAMA PARA CONOCER A
  DONDE APUNTA EL PUNTERO
3 ; DE PILA Y EL VALOR QUE
  SE OBTENDRA EN LA
  SIGUIENTE EXTRACCION DE
4 ; LA PILA

A410      10      ORG  42000
A410      20      ENT  42000
BB5A      30 PRIN  EQU  47962
A410      EI      60 PROG1 POP  HL
A411      E5      70      PUSH HL
A412      2234AB  80      LD   (43828),HL
A415      CD5AA4  70      CALL PMESS1
A418      CD22A4  J00      CALL PR0G2
A41B      ED7334AB 110     LD   (43828),SP
A41F      CD61A4  120     CALL PMESS2
A422      11F0D8  130 PROG2 LD   DE,-10000
A425      CD41A4  140     CALL REDN
A428      1118FC  150     LD   DE,-1000
A42B      CD41A4  160     CALL REDN
A42E      119CFF  170     LD   DE,-100
A431      CD41A4  180     CALL REDN
A434      1EF6    190     LD   E,-10
A436      CD41A4  200     CALL REDN
A439      3A34AB  210     LD   A,(43828)
A43C      F630    220     OR   #30

```

A43E	C35ABB	230		JP	PRIN
A441	3E30	240	REDN	LD	A, «30
A443	3C	250	FNUM	INC	A
A444	2A34AB	260		LD	HL, (43828)
A447	19	270		ADD	HL, DE
A44S	2234AB	280		LD	(43828), HL
A44B	3BF6	290		JR	'C, FNUM
A44D	2A34AB	300		LD	HL, (43828)
A450	ED52	310		SBC	HL, DE
A452	2234AB	320		LD	(43828), HL
A455	3D	330		DEC	A
A456	CD5ABB	340		CALL	PRIN
A459	C9	350		RET	
A45A	0607	360	PMESS1	LD	B, 7
A45C	216EA4	370		LD	HL, MESS1
A45F	1805	380		JR	MLOOP
A461	0604	390	PMESS2	LD	B, 4
A463	2175A4	400		LD	HL, MESS2
A466	7E	410	MLOOP	LD	A, (HL)
A467	CD5ABB	420		CALL	PRIN
A46A	23	430		INC	HL
A46B	10F9	440		DJNZ	MLOOP
A46D	C9	450		RET	
A46E	0A0D	460	MESS1	DEFW	#0D0A
A470	285350293D	470		DEFM	" (5P) ="
A475	2053503D	480	MESS2	DEFM	" SP ="

Pasa 2 errors: 00

Table used: 132 from 196  
 Executes: 42000

Sumas de comprobación: 0581, 05B6, 0561, 0580, 04F9, 02C7, 047C, 0403,  
 03A9, 0300, 013D

### Figura 9.!

No necesitaremos explicar muchas cosas del programa, ya que, en su mayor parte, le será familiar.

Se ha cambiado algo la forma de imprimir los números. Para conseguir a partir de una cifra su código ASCII se carga con #30 el acumulador desde el comienzo, excepto para la última cifra, pues en este caso se carga la cifra y luego se utiliza la instrucción OR.

La instrucción de la línea 110 es nueva, pero es fácilmente comprensible a través de su código nemotécnico por ser similar al de instrucciones ya explicadas: las del tipo LD (nn), rr. Ahora, sin embargo, en lugar de un par de registros se emplea el registro SP de 16 bits. Además, el código de LD (nn), SP es 1110 1101 01 110 011 n n, completamente análogo a los de la figura 5.7 utilizando 11b como código de 2 bits para SP. Cabe preguntarse qué representa el código 10b en este tipo de instrucciones. Parece lógico que represente a HL como en otros casos y de hecho así ocurre, si bien las instrucciones LD HL, (nn) y LD (nn), HL tienen además otros códigos más bre-

ves que ya explicamos. Puede comprobar que este otro código funciona también, sustituyendo la línea 80 (de 3 bytes) por las cuatro líneas

```
80 DEFB #ED
81 DEFB %01100011
82 DEFB #34
83 DEFB #AB
```

volviendo a ensamblar el programa y observando que el programa sigue funcionando exactamente igual.

El programa comienza por extraer el valor (de 2 bytes) que hay en el extremo de la pila y lo carga en HL; a continuación devuelve este valor a la pila para dejarla inalterada, pero HL guarda ya una copia de dicho valor. HL se carga en la posición de memoria 43828. Luego, la rutina PMESS1 imprime el mensaje '(SP) = ' y a continuación la rutina PROG2 se encarga de imprimir el valor del extremo de la pila. En este momento se llevan realizados dos CALL y dos RET, por lo que el puntero de pila estará como al comienzo del programa. El contenido de SP se deposita ahora en memoria para ser impreso después. Previamente la rutina PMESS2 imprime el mensaje ' SP = ' Inmediatamente se entra en la rutina PROG2, que es la que imprime el número. Como se ha accedido a esta rutina sin un CALL, la instrucción RET final provocará la vuelta a BASIC o al ensamblador, según el caso.

Puede usted comprobar cómo se puede manipular la pila intencionadamente cargando las siguientes líneas previas al programa anterior:

A409		5	ORG	41993
A409		6	ENT	41993
A409	2110A4	7	LD	HL, PROG1
A40C	E5	8	PUSH	HL
A40D	E5	9	PUSH	HL
A40E	E5	10	PUSH	HL
A40F	E3	20	PUSH	HL

Si se utiliza el CARGADOR HEX la dirección de HIMEM debe ser 41992 y la dirección inicial 41993. Vuelva a ensamblar el programa y ejecútelo. Si ha utilizado nuestro cargador comience la ejecución en A409h (41993).

Lo que hace ahora el programa es ejecutarse cinco veces. La culpa de los cuatro retornos suplementarios es de las cuatro instrucciones PUSH, que hacen que el RET pase el control a la dirección de PROG1 en lugar de volver al BASIC o al ensamblador.

Las instrucciones que hemos visto son las únicas que modifican implícitamente el puntero de pila cada vez que se las ejecuta. Pero hay otra serie de instrucciones que hacen posible la manipulación de la pila, pasando información desde y hacia la pila.



Un primer grupo de instrucciones está formado por las instrucciones de carga que afectan al puntero de pila, SP; son las instrucciones más directas. Al encender el Amstrad CPC464, el programa de arranque en frío del que ya hemos hablado inicializa el puntero de pila en una dirección alta, la 49144 (BFF8h), desde donde irá creciendo hacia abajo. Normalmente no hará falta modificar esta dirección de la base de la pila, pero otras veces puede ser conveniente alterar la posición de la pila modificando el contenido de su puntero SP.

*Mantenga siempre el puntero de pila apuntado hacia una dirección par, sobre todo en el Amstrad, donde puede intercambiarse áreas de memoria. En caso contrario puede llegar a ocurrir que quede desactivada la mitad de un valor almacenado, permaneciendo el byte restante en la pila. Lo mejor es inicializar el puntero en una dirección que sea múltiplo de 256, ya que esto permitirá el máximo crecimiento de la pila antes de cambiar de página de memoria.*

Existen para SP las instrucciones de carga que ya hemos visto para los pares de registros. Los códigos de estas instrucciones se forman según las reglas que ya explicamos, teniendo en cuenta que el código de 2 bits para SP es 11. Estos códigos son:

ENSAMBLADOR		HEX	BINARIO							
LD	SP, nn	31 n n	00	110	001	n	n			
LD	SP, (nn)	ED 76 n n	11	101	101	01	111	011	n	n
LD	(nn), SP	ED 73 n n	11	101	101	01	110	011	n	n

Como hemos dicho, hay ocasiones en que es necesario, o simplemente conveniente, cambiar el puntero de pila. Así ocurre, por ejemplo, cuando hay alguna instrucción prioritaria sobre cualquier cosa se esté realizando. En ese caso puede no existir la posibilidad de asegurarse de que la pila va a quedar equilibrada y, por lo tanto, debe inicializarse la pila en una dirección conocida.

Un buen ejemplo de situación en que es provechoso alterar el puntero de pila, lo da el programa de la figura 9.2. En este caso se almacena el valor de SP en memoria al comenzar el programa, para recuperarlo al final.

El programa es una modificación del de la figura 8.3, utilizando la instrucción PUSH; se emplea así menos tiempo en rellenar la pantalla que de la manera original. En este programa se carga en SP el valor 0. Como la dirección por debajo de 0 es -1, o sea FFFFh, la pila comienza a ocupar la parte superior del área de memoria reservada a la pantalla a medida que se ejecutan las instrucciones PUSH. EN HL se carga el valor 5C5Ch, que es el mismo

Hisoft GENA3 Assembler. Page

Pass 1 errors: 00

		1	;	FIG 9,2	
		2	;	RELLENO DE LA PANTALLA	
88B8		10		ORG	35000
8BBB		20		ENT	35000
88BB	ED73D188	30		LD	(SPWD),SP
88BC	310000	10		LD	SP,#0
88BF	215C5C	50		LD	HL,#5C5C
88C2	0E20	60		LD	C,#20
88C4	0600	70	BLOOP	LD	B,#0
88C6	E5	80	SLOOP	PUSH	HL
88C7	10FD	90		DJNZ	SLOOP
BBC9	OD	100		DEC	C
88CA	20F8	110		JR	NZ,BLOOP
88CC	ED7BD1B8	120		LD	SP,(SPWD)
88D0	C9	130		RET	
88D1	0000	140	SPWD	DEFW	0

Pass 2 errors: 00

Table used: 48 from 127

Executes: 35000

Sumas de comprobación: 03C3, 034B, 03BA

Figura 9.2

con que se cargaba A en el programa de la figura S.3 pero repetido dos veces; ahora se llenarán cada vez dos posiciones de memoria.

Luego viene el núcleo del programa, que es un doble bucle anidado. Es una técnica muy corriente para superar las limitaciones de los valores que pueden almacenar los contadores. El bucle externo, BLOOP, pasa 32 veces; en cada una de ellas se ejecuta 256 veces el bucle interno, SLOOP. La instrucción PUSH HL se ejecuta entonces  $32 \times 256 = 8192$  veces y, como cada vez se rellenan dos posiciones, se llena un total de 16384 (4000h) bytes. El programa termina recuperando el valor inicial de SP y ejecutando un RET.

El puntero de pila, SP, se puede utilizar también en las operaciones aritméticas de 16 bits. Se emplea en ADD, ADC, SBC, INC y DEC del mismo modo que los pares de registros. Los códigos binarios de las instrucciones se forman de la misma manera, pero utilizando 11 en los bits 5 y 4 en el caso de SP. Por ejemplo,

ADD HL,DE	es	00 011 001	luego	ADD HL,SP	es	111 001
DEC BC	es	00 001 011	luego	DEC SP	es	111 011

La siguiente instrucción permite intercambiar entre el valor del extremo de la pila con el contenido de HL. Como se trata de un intercambio (*exchange*), el código nemotécnico de la instrucción será EX; esto se completará con (SP) y HL, que son las dos cosas que se intercambian. Los códigos completos son:

ENSAMBLADOR	HEX	BINARIO
EX (SP),HL	E3	11 100 011

Es una de las instrucciones referentes a la pila que se utiliza más; se emplea para cambiar la dirección de vuelta de una subrutina desde la propia subrutina, o incluso para añadir subrutinas adicionales.

Supongamos por ejemplo que tenemos una subrutina cuya finalidad es realizar ciertos cálculos de 16 bits para el programa principal. Cada resultado se almacena en HL, como ya sabemos. Si hay varios cálculos que hacer, será preciso liberar HL para realizar otro de los cálculos. Luego habrá que guardar en memoria el contenido de HL para que lo recupere más tarde el programa principal. La instrucción LD (nn),HL puede servir, pero emplea 3 bytes y otros 3 la instrucción que devuelve el valor a HL. Lo más económico es almacenar el resultado en la pila, pero, si se hace directamente, se imposibilita la extracción de la dirección de retorno de la subrutina. Lo que se puede hacer entonces es almacenar el valor, pero de manera que intercambie su posición con la dirección de la vuelta al programa. Esto se consigue con las dos instrucciones

EX (SP),HL y PUSH HL

La primera almacena el resultado numérico y extrae la dirección de vuelta; la segunda coloca de nuevo en la pila la dirección de vuelta. Se utilizan así 2 bytes, y otro más cuando el programa principal recupere el resultado.

Hay por fin una última instrucción. Es un poco rara para lo que hemos visto hasta ahora, ya que permite cargar un registro de 16 bits con el contenido de otro. Se trata de

ENSAMBLADOR	HEX	BINARIO
LD SP,HL	F9	11 111 001

que se utiliza cuando una dirección de vuelta proviene del resultado de un cálculo

Aquí termina nuestra explicación, que puede haberle resultado pesada. Ahora debe usted mismo experimentar con los ejemplos que hemos dado,

cargándolos y ejecutándolos en su Amstrad. No se olvide de grabar el programa antes de ejecutarlo; si algo sale mal podrá desconectar y volver a encender el ordenador, y tendrá el programa a su disposición para corregirlo. Vigile siempre que haya el mismo número de PUSH que de POP, y que cada CALL lleve aparejado un RET.

## Resumen

Vamos a resumir las instrucciones explicadas en este capítulo. Utilizaremos los símbolos:

- r = cualquiera de los registros de 8 bits (A, B, C, D, E, H o L)
- rr = cualquier par de registros que se utilicen como uno de 16 bits
- n = un número de 8 bits, o sea, entre 0 y 255
- nn = un número de 16 bits, o sea, entre 0 y 65535
- ( ) rodeando un número o un par de registros = el contenido de la dirección.
- PC = contador de programa
- SP = puntero de pila

La pila va creciendo hacia posiciones más bajas de la memoria. Su extremo es la dirección más baja de las que ocupa la pila; a él apunta SP.

PUSH coloca en el extremo de la pila el contenido de un par de registros, y actualiza SP para que apunte al nuevo extremo.

POP hace justamente lo contrario.

Todo rr habitual y el par AF pueden ser utilizados con PUSH y POP.

Todas las instrucciones de carga y aritméticas de 16 bits, así como INC y DEC, pueden utilizar SP.

EX (SP),HL intercambia el contenido del extremo de la pila con el contenido de HL.

Cada PUSH debe ir acompañado del correspondiente POP. En la instrucción POP se puede utilizar un rr diferente del empleado en PUSH.

Cada CALL debe llevar el correspondiente RET.

## Instrucciones que trabajan con un solo bit

Entre los aspectos particulares que distinguen al Z80 de otros microprocesadores de 8 bits está el hecho de poseer instrucciones que trabajan con un solo bit. Con estas instrucciones se puede poner a 1 o ponerse a 0 un bit cualquiera de un registro o de una posición de memoria (sin alterar los demás bits), y también se puede averiguar el estado de un bit determinado.

Cabe preguntarse si son verdaderamente necesarias estas instrucciones, ya que todos esos resultados se pueden obtener mediante otras.

Por ejemplo, podemos trabajar con el bit 5 de A de la manera siguiente: para poner a 1 el bit basta utilizar

```
OR %00100000
```

para poner a 0 el bit basta utilizar

```
AND %11011111
```

y, finalmente, la instrucción

```
AND %00100000
```

activará el indicador de cero si el bit es 0 y desactivará el indicador de cero si el bit es 1.

Claro que todo esto supone que el bit con el que se trabaja es un bit del acumulador. Si no es así, las cosas son un poco más costosas. Vamos a ver que habría que hacer para poner a 0 el bit 5 de una posición de memoria que representaremos, por ejemplo, por 'tb'. La secuencia de operaciones sería la siguiente:

- |                                   |               |
|-----------------------------------|---------------|
| 1) Guardar el contenido de A      | PUSH AF       |
| 2) Cargar el byte en A            | LD A,(tb)     |
| 3) Poner a 0 el bit 5             | AND %11011111 |
| 4) Devolver el byte a su posición | LD (tb),A     |
| 5) Recuperar el contenido de A    | POP AF        |

Se requieren, pues, 10 bytes de programa para una operación tan sencilla.

Este número se puede reducir algo si se utiliza HL como puntero de la manera siguiente:

```
PUSH AF
LD HL,tb
LD A,(HL)
AND %11011111
LD (HL),A
POP AF
```

Así se reduce el programa a 9 bytes, lo que no representa un gran ahorro.

Para comprobar cuál es el valor de un bit hay que variar el procedimiento, ya que la operación se basa en el examen del indicador de 0 y, al emplear POP AF, los indicadores recuperan el estado que tenían antes del programa. El almacenamiento de A se puede hacer en otra posición de memoria que denotaremos por 'sb'. El programa para comprobar el valor del bit 5 de la posición de memoria tb sería el siguiente:

```
LD (sb),A
LD HL,tb
LD A,(HL)
AND %00100000
LD A,(sb)
```

El programa ocupa 12 bytes.

El objeto de desarrollar estos programas, que van a ser completamente inútiles, es demostrar la conveniencia de disponer de operaciones directas para tales tareas.

Las instrucciones que sirven para poner a 1 y a 0 un bit tienen por código SET y RES respectivamente. El bit puede ser de un registro de uso general, de A o de la posición de memoria apuntada por HL. Sus códigos binarios son

ENSAMBLADOR	BINARIO					
SET b,r	11	001	011	11	b	r
RES b,r	11	001	011	10	b	r

donde r se debe sustituir por el código usual de 3 bits, o sea, 000 para B, . . . ,110 para (HL) y 111 para A. También b debe ser sustituido por el número del bit que se deba alterar, o sea, b puede ser desde 000 para el bit 0 (el menos significativo) hasta 111 para el bit 7 (el más significativo).

Obsérvese que las instrucciones que trabajan con un bit ocupan 2 bytes, de los que el primero es siempre 11 001 011 (CBh).

Por ejemplo, las instrucciones para poner a 1 el bit 5 del registro B y para poner a 0 el bit 3 de la posición de memoria apuntada por HL son

ENSAMBLADOR	HEX	BINARIO							
SET 5,B	CB E8	11	001	0H	11	101	000		
RES 3, (HL)	CB 9E	11	001	011	10	011	110		

Las instrucciones SET y RES no afectan a ningún indicador.

La instrucción que sirve para comprobar cuál es el estado de un bit tiene por código nemotécnico BIT, y su código binario es similar a los precedentes:

ENSAMBLADOR	BINARIO							
BIT b, r	11	001	011	01	b	r		

Por ejemplo, la instrucción para comprobar el bit 2 del registro H es:

ENSAMBLADOR	DECIMAL	BINARIO							
BIT 2,H	C8 54	11	001	011	01	010	100		

Pero, ¿de qué manera nos dice la instrucción BIT cuál es el valor del bit? Nos lo dice mediante el indicador de cero. Al ejecutar la instrucción BIT, el indicador de cero se pondrá a 1 si el bit vale 0, y se pondrá a 0 si el bit vale 1. La instrucción BIT no afecta al indicador de arrastre, pero los otros indicadores, aparte del de cero, pueden verse afectados de manera imprevisible.

Uno de los campos en que son útiles las instrucciones que trabajan con un bit es el de la codificación de informaciones; sobre todo cuando se trata de información alternativa que puede darse con un 'si' o un 'no'. Representando 'si' por un 1 y 'no' por un 0, cada una de las informaciones ocupará un bit. Así, por ejemplo, consideremos los siguientes datos alternativos sobre cada empleado de una fábrica (que ponemos también en inglés para ayudarle a comprender el programa que introduciremos más adelante):

1) Male/Female	Hombre/Mujer
2) Married/Single	Casado/Soltero
3) Children/Childless	Con niños/Sin niños

4) Driving licence/No driving licence	Permiso de conducir/No permiso
5) Salaried/Hourly paid	Salario/Por horas
6) Key holder/Not key holder	Tiene llave/No tiene llave
7) Security cleared/Not Security cleared	Seguridad comprobada/Dudosa seguridad

Todos estos datos se pueden almacenar en siete bits de un byte, dejando el bit restante para indicar si el byte está o no en uso.

Hisoft GENA3 Assembler. Page 1.

Pass 1 errors: 00

```

10 ; FIG 10.1 - PROGRAMA QUE MUESTRA
20 ; LAS DIFICULTADES DE MANEJAR
30 ; REGISTROS CON OR Y AND
88B8 40 ORG 35000
88B8 50 ENT 35000
BB5A 60 PRINT EQU 47962
881B 70 GETKEY EQU 47896
88B8 21438A 80 LD HL, FREE
B8BB CD5689 90 NXTREC CALL CRLF
8BBE CD5689 100 CALL CRLF
B8C1 0609 110 LD B, 9
88C3 CDFBB8 120 CALL PR_MSG
B8C6 CD6389 130 CALL KEYIN
BBC7 FE66 140 CP "y"
8BCB 2B4E 150 JR Z, LSTREC
88CD 3E01 160 LD A, #1
88CF 77 170 LD (HL), A
88D0 0607 180 LD B, 7
BBD2 0E02 190 LD C, 2
BBD4 CD5689 200 NXTBIT CALL CRLF
B8D7 CDFBB8 210 CALL PR_MSG
88DA C5 220 PUSH BC
88DB 060A 230 LD B, 10
88DD CDF888 240 CALL PR_MSG
8BE0 C1 250 POP BC
88E1 CD6389 260 CALL KEYIN
8BE4 FE79 270 CP "y"
88E6 2005 280 JR NZ, NO
88E8 7E 290 LD A, (HL), A
88E9 B1 300 OR C
88EA 77 310 LD (HL), A
88EB 1B06 320 JR SLA
88ED FE6E 330 NO CP "n"
88EF 2802 340 JR Z, SLA
BBF1 1BE1 350 JR NXTBIT
88F3 79 360 SLA LD A, C
88F4 B7 370 ADD A, A

```



88F5	4F	380	LD	C,fl
88F6	10DC	390	DJNZ	NXTBIT
88F8	23	400	INC	HL
88F9	18C0	410	JR	NXTREC
88FB	CD6C89	420	PR_MSG	CALL SAVREG
88FE	217689	430	LD	HL,MSG T
8901	CB7E	440	FNDMSG	BIT 7, <NL>
B903	23	450	INC	HL
B904	28FB	460	JR	Z, FNDMSG
B906	10F9	470	DJNZ	FNDMSG
8908	CD0F89	480	CALL	NXTCHR
890B	CD7189	490	CALL	RESREG
B90E	C9	500	RET	
B90F	7E	510	NXTCHR	LD A, (HL)
8910	E67F	520	AND	%011111
8912	CD3ABB	530	CALL	PRINT
B915	C87E	540	BIT	7, (HL)
B917	C0	550	RET	NZ
B918	23	560	INC	HL
8919	18F4	570	JR	NXTCHR
B91B	21438A	580	LSTREC	LD HL, FREE
891E	CD56B9	590	CALL	CRLF
8921	CD56B9	600	CALL	CRLF
8924	0608	610	LD	B, 8
B926	CDFB88	620	CALL	PR_MSG
B929	0601	630	PR_REC	LD B, 1
892B	E5	640	PUSH	HL
892C	CD56B9	650	CALL	CRLF
B92F	CD56B9	660	CALL	CRLF
B932	CD18BB	670	CALL	GETKEY
8935	E1	680	POP	HL
8936	7E	690	LD	A, (HL)
B937	23	700	INC	HL
B938	A7	710	AND	A
B939	C8	720	RET	Z
893f1	87	730	P_ITEM	ADD A,A
B93B	CDFB88	740	CALL	PR_MSG
B93E	F5	750	PUSH	AF
893F	3007	760	JR	NC, NOT
8941	3E59	770	LD	A, "Y"
8943	CD5ABB	780	CALL	PRINT
B946	1B05	790	JR	NXTITM
B94B	3E4E	800	NOT	LD A, "N"
894A	CD5ABB	810	CALL	PRINT
B94D	04	820	NXTITM	INC B
B94E	CD5689	830	CALL	CRLF
8951	F1	840	POP	AF
8952	28D5	850	JR	Z, PR RE (
B954	1BE4	860	JR	P_ITEM
B956	F5	870	CRLF	PUSH AF
8957	3E0D	880	LD	A, #0D
B959	CD5ABB	890	CALL	PRINT
B95C	3E0A	900	LD	A, #0A
895E	CD5ABB	910	CALL	PRINT
8961	F1	920	POP	AF

8962	C9	930	RET	
8963	CD188B	940	KEYIN CALL	GETKEY
8966	CD5ABB	950	CALL	PRINT
8969	F620	968	OR	#20
896B	C9	970	RET	
B96C	E3	980	SAVREG EX	(SP),HL
8960	C5	99?	PUSH	BC
896E	F5	1000	PUSH	
896F	E5	1010	PUSH	AF
8970	C9	1020		
8971	E1	1030	RESREG POP	HL
8972	F1	1040	POP	AF
B973	C1	1050	POP	BC
8974	E3	1060	EX	(SP),HL
8975	C9	1070	RET	
8976	A0	1080	MSGTBL DEFB	#A0
8977	53154355	1090	DEFM	"SECURITY C
B9B8	A0	1100	DEFB	#A0
8989	4B455920	1110	DEFM	"KEY HOLDER
899A	A0	1120	DEFB	#A8
899B	53414C41	1130	DEFM	"SALARIED ?
89AC	A0	1140	DEFB	#A0

## Hisoft GENA3 AssemLer. Page

B9AD	44E34956	1150		"DRIVING LICENCE ?"
89BE	A0	1160		
89BF	4620544F	1170	DEFB	
B9D0	A0	1180	DEFB	#A0
B9D1	4D415252	1190	DEFM	"MARRIED ?
89E2	A0	1200	DEFB	#A0
89E3	4D414C45	1210	DEFM	"MALE ?
89F4	A0	1220	DEFB	
89F5	0A0A	1230	DEFB	
89F7	464F5220	1240	DEFM	"FOR NEXT RECORD PRESS AN
8A14	0788	1250	DEFW	#8807
BA16	4620544F	1260	DEFM	"F TO FINISH OR ANY OTHER
SA32	20544A20	1270	DEFM	" TO GO ON"
SA3B	07A0	1280	DEFW	#A007
BA3D	20592F4E	1290	DEFM	" Y/N
8A41	A0A0	1300	DEFW	#A0A0
8A43	0000	1310	FREE DEFW	#0000

Pass2 errors: 00

Table used: 257 from 327  
 Executes: 35000

Figura 10.1

Para crear registros de este tipo pueden servir perfectamente las instrucciones AND y OR. Pero con ellas es verdaderamente complicado cambiar un bit específico de un registro que ya está lleno. Para estos aspectos es preferible emplear las instrucciones que alteran un bit.

El programa de la figura 10.1 le ayudará a comprender estas dificultades. No le sugerimos que lo introduzca ahora, pero puede hacerlo si quiere para ver qué ocurre. El programa se debe cargar con un ensamblador; si usted lo carga utilizando el código hexadecimal, debe advertir que el código de los mensajes (líneas 1080 y siguientes) no está completo en ninguno de ellos, puesto que en el listado aparecen sólo los 4 primeros bytes de cada uno.

El programa carga registros con las siete características de las que hemos hablado antes; usted tendrá que introducir los datos pulsando 'Y' para 'sí' y 'N' para 'no'. Mediante el mensaje 'F TO FINISH OR ANY OTHER KEY TO GO ON' el programa le pedirá si desea que los registros ya cargados se impriman en la pantalla (pulse 'F' para esta opción) o si desea cargar nuevos registros (pulse cualquier otra tecla). Cuando se imprimen los registros en la pantalla, la impresión se detiene en cada registro, y el mensaje 'FOR NEXT RECORD PRESS ANY KEY' le recuerda que debe pulsar una tecla para pasar al siguiente.

Se utilizan muchas de las técnicas e instrucciones que ya hemos comentado, y también algunos trucos. Trate de ver por qué aparece la letra 'Y' después del mensaje 'FOR NEXT ...' de la línea 1240.

Para poner a 1 el bit correspondiente cuando la respuesta es 'Y', se emplea la instrucción lógica OR C con un byte C que contiene un 1 en la posición correspondiente. Como la posición del 1 debe ir variando, la subrutina SLA emplea la instrucción ADD A,A para multiplicar por 2 el byte precedente, lo que equivale a desplazar el 1. El mismo artificio se emplea para poner a 1 el indicador de arrastre cuando, en la impresión de los registros, se llega a una cuestión que ha sido respondida con 'Y'.



## Rotaciones y desplazamientos

En las últimas consideraciones que hicimos en el capítulo precedente acerca del programa de la figura 10.1, vimos que, para desplazar hacia la izquierda todos los bits de un byte, lo que hay que hacer es multiplicar por 2 el valor del byte. En ese programa la multiplicación por 2 se llevaba a cabo sumando el byte consigo mismo. La sección del programa que se ocupaba de esta tarea llevaba la etiqueta SLA; el propósito de esta etiqueta es hacer notar que la subrutina en cuestión realiza el mismo trabajo que una de las instrucciones que veremos ahora: la que realiza el *desplazamiento aritmético a la izquierda* (o *Shift Left Arithmetic*), que se denota por SLA.

En resumen, si un número binario se suma consigo mismo o, lo que es igual, se multiplica por 2, el efecto es desplazar el número una posición hacia la izquierda. Este efecto de la multiplicación no es específico del sistema binario (sí el de la suma). Si se multiplica un número escrito en un sistema de numeración por la base del sistema, el efecto es desplazar el número a la izquierda. Por ejemplo:

en binario	$1010110b * 10b = 10101100b$	(10b es 2 en decimal)
en decimal	$1234567 * 10 = 12345670$	
en hexadecimal	$789ABCDh * 10h = 789ABCD0h$	(10h es 16 en decimal)

Volviendo al programa, se observa que el hecho de tener que utilizar constantemente el acumulador para desplazar un byte no es cómodo ni conveniente.

Otro de los problemas es no poder hacer lo mismo para provocar un desplazamiento a la derecha, pues de esa manera se podría presentar la información en el mismo orden en que fue introducida. Lo que hace el programa es utilizar el mismo procedimiento que antes para ir activando el indicador de arrastre cada vez que un dato archivado es 1. Se podría modificar el programa de varias maneras para que imprimiese la información en el mismo orden de introducción. Por ejemplo, desplazando el byte en A a continuación de la instrucción OR en lugar de actuar sobre el registro C. Pero esto traería nuevos problemas, puesto que el desplazamiento debería hacerse

también en caso de respuesta negativa y, en ese caso, el programa se bifurca antes de la instrucción OR.

Lo que parece en todo caso necesario es disponer de un conjunto de instrucciones de desplazamiento, y esto no sólo por los inconvenientes que hemos señalado, sino también para poder realizar divisiones de una manera sencilla.

Ya hemos dicho que, cuando un número se multiplica por la base del sistema de numeración en que está escrito, se desplaza una posición hacia la izquierda. Pero, ¿qué sucede cuando se lo divide por la base? En ese caso se desplaza una posición hacia la derecha y la cifra de la derecha sale fuera (a la zona de los números fraccionarios). En el caso de un byte, el número de la derecha debe desaparecer; veremos que se lo puede recuperar en el indicador de arrastre.

El Z80 dispone de instrucciones para desplazar un byte a la izquierda o a la derecha. Comenzaremos por explicar el desplazamiento a la izquierda.

El desplazamiento a la izquierda tiene como código SLA (ya hemos explicado que proviene de *Shift Left Arithmetic*). Realiza la misma operación que ADD A,A, pero puede utilizar, además de A, los registros de uso general y (HL). Su código binario se compone de 2 bytes; el primero es el prefijo CBh, que se emplea para los desplazamientos y las rotaciones, así como para las instrucciones que trabajan con un bit, como hemos visto en el capítulo anterior. El código completo es

ENSAMBLADOR	HEX	BINARIO
SLA r	CB 20-27	11 001 011 00 100 r

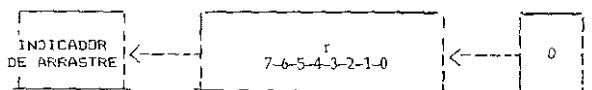


Figura 11.1

donde hay que sustituir r por el código de 3 bits que se emplea para los registros de uso general, A y (HL).

En algunas ocasiones no tiene ninguna importancia el hecho de que la instrucción SLA expulse del byte el bit 7; pero otras veces, sobre todo en las operaciones de multiplicación, este bit es fundamental, pues es el más significativo. Afortunadamente, este bit se guarda en el indicador de arrastre ya que se activará justamente cuando el bit 7 sea 1. En el programa de la figura 6.8 vimos cómo se recuperaba el arrastre en una suma, incorporándolo al

siguiente byte con ADC; es exactamente lo que hay que hacer cuando se suman números sin signo. Veamos qué técnica hay que emplear en la multiplicación. Para multiplicar por 2 el contenido de A se puede utilizar el programa

```

MULT      SIA  A
          LD   (RESULT),A
          LD   A,(RESULT+1)
          ADC  A,A
          LD   (RESULT+1),A
          RET
RESULT    DEFW 0

```

Figura 11.2

El resultado de la multiplicación queda almacenado en las posiciones RESULT y RESULT+1, con el byte más significativo en RESULT+1, o sea, en la forma habitual de almacenamiento de un número de 16 bits.

Si se usa repetidamente esta rutina, puede servir para multiplicar por una potencia de 2. Por ejemplo:

```

LD  A, 1
CALL MULT ;   en RESULT hay ahora 2
LD  A, (RESULT)
CALL MULT ;   en RESULT hay ahora 4
LD  A, (RESULT)
CALL MULT ;   en RESULT hay ahora 8

```

Figura 11.3

y así sucesivamente. El programa funcionará hasta que el resultado exceda de 65535, o sea, hasta que se realicen 16 llamadas a la rutina; además, el indicador de arrastre quedará entonces a 1. El programa no es bueno, ni mucho menos, pero ilustra el empleo del desplazamiento a la izquierda para

multiplicar. Cuando se va a multiplicar un número negativo con esta técnica, el byte más significativo de RESULT se debe cargar con 1111111b antes de comenzar los cálculos; si no, el resultado final sería positivo. No nos ocuparemos ahora de mejorar el programa, sino que pasaremos a explicar el desplazamiento a la derecha.

Hay dos tipos de desplazamiento a la derecha, que reciben los calificativos de lógico y aritmético.

El *desplazamiento lógico a la derecha* (o *Shift Right Logical*) tiene por nemotécnico SRL. A pesar de su nombre, es la instrucción que se corresponde con el desplazamiento aritmético a la izquierda. Su código y un esquema de su funcionamiento son los siguientes:

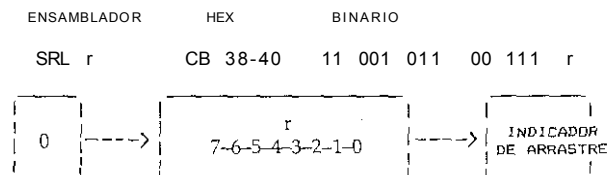


Figura 11.4

El código es similar al de SLA, con dos bytes, el primero de los cuales es CBh. En las instrucciones que veremos en este capítulo, lo que distingue una de otra son los bits 5, 4 y 3 del segundo byte, además, claro está, de los 3 bits que corresponden al código del registro.

A primera vista, esta instrucción parece que puede servir para transformar nuestra rutina de multiplicación en otra de división por 2; para ello bastaría con reemplazar SLA por SRL e invertir el orden de las operaciones, a fin de empezar por el byte más significativo. El problema fundamental es que no hay manera de utilizar el bit de arrastre que se origine en el byte más significativo para incorporarlo a la operación que se realice con el siguiente byte. Esto nos hace restringir la rutina a enteros de 8 bits, como muestra la figura 11.5.

Si al comienzo del programa A contiene el número 100 {64h 01100100b}, después de la ejecución la posición RESULT+1 almacenará 50(00110010b), que es lo correcto. Si se divide un número impar, el resto quedará en el indicador de arrastre. Así, si la rutina se emplea para 101 (65h 01100101b), el resultado en RESULT+1 será 50 y el indicador de arrastre quedará activado.

¿Qué sucede si se divide un número negativo (o sea, interpretado con sig-



```

DIVD    SRL    A
        LD     (RESULT+1),A
        RET
RESULT  DEFW   00
    
```

Figura 11.5

no)? Si nuestra rutina se emplea para -26 (E6h 11100110b), el resultado en RESULT+1 será 01110011b o 73h o 115 decimal, que es totalmente incorrecto. Así pues, la instrucción SRL no puede ser interpretada como desplazamiento aritmético, y por eso ha recibido otro calificativo.

El *desplazamiento aritmético a la derecha* (Shift Right Arithmetic) o SRA, lo que hace es preservar el bit de signo. Si en el ejemplo anterior sustituimos SRL por SRA, el resultado de la última operación será 11110011b o -13 o F3h, que es lo correcto. Los códigos y el esquema de funcionamiento para esta instrucción son:

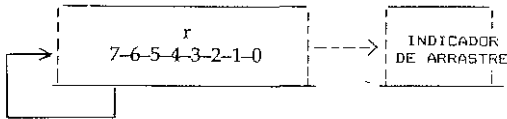


Figura 11.6

Ahora que conocemos los desplazamientos y, por lo tanto, la multiplicación y la división por 2, vamos a tratar de aprender a multiplicar y dividir por números diferentes de 2. Por el momento supondremos que todos los números empleados caben en un byte: así las cosas serán mas simples y podremos concentrarnos en comprender los principios de la multiplicación y la división, antes de entrar en cálculos más pesados. Para cálculos con números sin signo, esta suposición obliga a que el resultado de las multiplicaciones sea inferior a 256, y a que el dividendo y el divisor de las divisiones sean inferiores a 256.

Una multiplicación se puede realizar simplemente mediante un proceso que sume el multiplicando tantas veces como indique el multiplicador. Puede comprobar esto utilizando el programa de la figura 11.7, que realiza la multiplicación de los códigos de las dos teclas que usted pulse en el teclado.

```

Hisoft GENA3   Assembler.   Page           1.

Pass 1 errors: 00

10 ; FIG 11.7
20 ; MULTIPLICACIÓN DE 8 POR 8
    BITS CON RESULTADO DE 8 BITS,
30 ; METODO DE LA SUMA REPETIDA

A7F8          40          ORG 43000
A7F8          50          ENT 43000
8818          60 GETKEY EQU 47896
A7F8 CD1888   70          CALL GETKEY
A7FB 4F       80          LD C,A
A7FC CD1BBB   90          CALL GETKEY
A7FF 47       100         LD B,A
A800 AF       110         XOR A ;A SE PONE A 0
A801 81       120 ADLOOP ADD A,C
A802 10FD     130         DJNZ ADLOOP
AB04 3278AB   140         LD (43896),A
A807 C3B4AA   150         JP 43700

Pass 2 errors: 00

Table used:   72          221
Executes: 43000

```

Figura 11.7. Sumas de comprobación: 0506, 0483.

Este programa está preparado para ser añadido al programa de la figura 6.13, que servía para imprimir un número en forma decimal (observe la instrucción JP 43700).

Ejecute el programa con CALL 43000 o con el comando R del ensamblador. El programa quedará esperando y, cuando usted pulse dos teclas, imprimirá el resultado. La mayor parte de las teclas posee códigos demasiado altos para que su producto quepa en un byte; pero puede obtener códigos pequeños pulsando caracteres de control, es decir, manteniendo pulsada la tecla [CONTROL] y pulsando entonces otra tecla. Por ejemplo, el carácter [CONTROL]G es el código 7 (y proporciona un pitido) y el carácter [CONTROL]J es el código 10(0Ah); su producto dará 70 como respuesta. En el apéndice 3 de la Guía del usuario encontrará los códigos generados por las distintas teclas.

El método del programa de 11.7 trabaja perfectamente para las multiplicaciones que debe hacer, pero es verdaderamente rudimentario; el bucle mediante el que repite la suma puede tener que realizarse hasta 127 veces. Para operaciones de 16 bits podría tener que hacer hasta 32767 veces la operación en el peor de los casos (cuando se realiza  $2 \times 32767$  en este orden); incluso con

el convenio de introducir primero el mayor número podría tener que repetir 256 veces la operación.

Existe un método que en principio es mejor. Es el método que se aprende en la escuela, y consiste en desplazar y sumar los productos simples. Observe cómo es este método, tanto en binario como en decimal:

BINARIO	DECIMAL
00010011	19d
00001011 *	11d
10011	19
100110	17
0	
10011000	
11010001	209

En binario es muy sencillo: por cada cifra del multiplicador se desplaza a la izquierda el multiplicando; el multiplicando se suma si la cifra era un 1 y no se suma si era un 0. De esta manera se realizan a lo sumo tantas sumas como

```

Hisoft GENA3 Assembler. Page      1.

Pass 1 errors:

10 ; FIG 11,8
20 ; MULTIPLICACION DE 8 POR 8 BITS
    CON RESULTADO DE 8 BITS, METODO
30 ; DE DESPLAZAMIENTO Y SUMA

A7FB          40          ORG  43000
A7F8          50          ENT  43000
BB18          60 GETKEY EQU  47896
A7F8 CD18BB   70          CALL GETKEY
A7FB 4F       80          LO   C,A
A7FC CD18BB   90          CALL GETKEY
A7FF 47       100         LD   B,A
A800 AF       110         XOR  A ; A SE PONE A
A801 CB38     120 ADLOOP  SRL  B
AB03 3001     130         JR   NC,NOADD
AB05 81       140         ADD  A,C
A806 CB21     150 NOADD   SLA  C
A80B 20F7     160         JR   NZ,ADLOOP
A80A 3278AB   170         LD   (43896),A
A80D C3B4AA   180         JP   43700

Pass 2 errors:

Table used:      84   from   230
Executes: 43000

```

Figura 11.8. Sumas de comprobación: 0550, 0397, 02CC.

cifras tiene el multiplicador, aunque se ahorra una suma cada vez que una cifra es 0. Nótese que esta circunstancia, que es rara en el sistema decimal, es frecuente en el caso binario, pues las cifras son solamente 0 y 1. Por lo tanto, este método exige un máximo de 8 sumas para números de 8 bits, y de 16 para números de 16 bits.

El programa de la figura 11.8 utiliza este método para multiplicar, y se lo puede enlazar con el de 6.13 para imprimir el resultado. Después de los pasos iniciales y de poner A a 0, se comprueba cuánto vale el bit menos significativo del multiplicador. La comprobación se hace mediante el desplazamiento a la derecha SRL que coloca dicho bit en el indicador de arrastre. Si el bit es 1, se suma el multiplicando y luego se desplaza a la izquierda (etiqueta ADLOOP). Si el bit es 0, sólo se desplaza a la izquierda, sin sumar (etiqueta NOADD). Se comprueba si quedan bits en el multiplicador y, de ser así, el proceso se repite. Finalmente se enlaza con la rutina de impresión.

La división es análoga a la multiplicación pero con el inconveniente de que se puede prolongar indefinidamente sin ser nunca exacta. Ocurre como en el cálculo de  $\pi$  ( $\pi$ ), que no puede terminar nunca. De hecho, hay divisiones muy sencillas que dan un resultado periódico sin fin. La solución es calcular el cociente y el resto (el cociente es el número de veces que el divisor puede restarse del dividendo sin que dé un resultado negativo).

Para usted debería ser ya familiar el programa de división similar al de la figura 11.7. De hecho, hemos empleado una rutina de división de este tipo en todos los programas que servían para imprimir un número en decimal. El procedimiento consiste en restar el divisor del dividendo y repetir este proceso contando las veces que puede hacerse hasta que el resultado dé negativo; entonces se recupera la última resta (se suma el divisor) y el número que se obtiene actúa como dividendo en la siguiente división.

Lo que ahorra mucho trabajo en la multiplicación era la eficacia de la instrucción SLA de desplazamiento aritmético a la izquierda, que además permitía trabajar con números de cualquier tamaño. Esto se debía a que dicha instrucción sacaba el bit de arrastre al indicador y a que el bit de arrastre podría incorporarse al bit menos significativo del byte siguiente. El proceso mezclaba una instrucción SLA y otra ADC en la forma siguiente:

	byte mas sign.	arr.	byte menos sig.
	7-6-5-4-3-2-1-0		7-6-5-4-3-2-1-0
	0 0 0 0 0 0 0 0	0	1 0 1 1 0 1 0 0
SLA menos	0 0 0 0 0 0 0 0	1	0 1 1 0 1 0 0 0
ADC mas,mas	0 0 0 0 0 0 0 1	0	0 1 1 0 1 0 0 0

Figura 11.9

Se tiene la suerte de que la instrucción ADC permite realizar un desplazamiento a la izquierda del bit de arrastre. Hay que pensar también en que todo esto se podría haber hecho mediante las instrucciones ADD HL,HL o ADC HL,HL, simulando así un desplazamiento a la izquierda de 16 bits.

Si se desea una división más eficaz, realizada mediante desplazamientos y restas en lugar de restas repetidas, son necesarias nuevas instrucciones de desplazamiento que permitan incorporar el bit de arrastre al bit más significativo. Naturalmente, hay muchas otras razones para justificar las nuevas instrucciones de desplazamiento.

El Z80 dispone de un amplio catálogo de instrucciones de desplazamiento que permiten la incorporación del bit de arrastre independientemente de los acumuladores (A para 8 bits y HL para 16). Todas estas instrucciones utilizan el indicador de arrastre, tanto para recibir el bit desplazado como para proporcionar el bit que rellene la posición liberada. Unas toman el bit del indicador de arrastre antes de introducir en él el bit desplazado. Otras colocan el bit desplazado en el indicador de arrastre, antes de introducir este indicador en la posición liberada. Todas ellas efectúan una rotación, ya sea a través del indicador de arrastre o incluyendo este indicador.

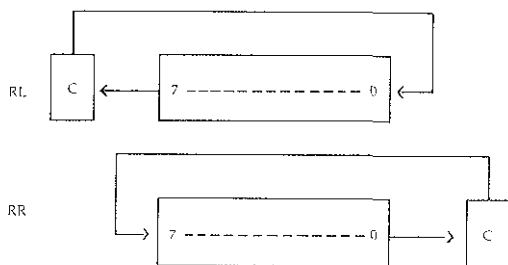


Figura 11.10

Todas las *rotaciones* comienzan por la letra R; luego llevan la letra L (de *left*), que indica izquierda, o la letra R (de *right*), que indica derecha, para señalar el sentido de la rotación. Se tienen así las instrucciones RL de *rotación izquierda* y RR de *rotación derecha* cuyo efecto se puede observar en la figura 11.10. Realizan una rotación de 9 bits; el bit desplazado pasa al indicador de arrastre, y el indicador de arrastre previo ocupa el lugar liberado.

Otras rotaciones son un poco diferentes y se llaman circulares: son la *rota-*

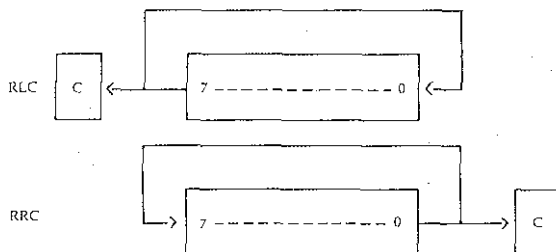


Figura 11.11

*ción circular izquierda*, o RLC, y la *rotación circular derecha*, o RRC. Su efecto se puede observar en la figura 11.11. Realizan una rotación de 8 bits; el bit desplazado pasa a la posición liberada, pero queda una copia de este bit en el indicador de arrastre.

El acumulador A posee, como los otros registros, estas cuatro rotaciones, pero además tiene otras específicas con la ventaja de que su código ocupa sólo un byte. Por lo demás se comportan como las anteriores salvo en cómo afectan a los indicadores Z, S y P/V).

Los códigos de todas estas rotaciones son:

ENSAMBLADOR	HEX	BINARIO
RL    r	CB 10 -- 17	11 001 011 00 010 r
RLA	17	00 010 111
RR    r	CB 18 -- 1F	11 001 011 00 011 r
RRA	1F	00 011 111
RLC    r	CB 00 -- 07	11 001 011 00 000 r
RLCA	07	00 000 111
RRC    r	CB 08 -- 0F	11 001 011 00 001 r
RRCA	0F	00 001 111

Con este conjunto de instrucciones, la puerta a una división rápida queda abierta. En las divisiones que realizábamos en los programas que imprimían un número en decimal se jugaba con dos ventajas. En primer lugar, el cociente nunca podría pasar de 9; por lo tanto no se empleaba demasiado tiempo en hacer las restas. Además, los divisores eran conocidos, pues eran siem-

Hisoft GENA3 Assembler. Page

Pass 1 errors:

```

10 ; FIG. 11.12 - DIVISION POR 2
      USANDO DESPLAZAMIENTO Y ROTACION

A7FB      20      ORG      43000
A7F8      30      ENT      43000
BB1G      40      GETKEY   EQU      47896
BB5A      50      PRINT    EQU      47962
A7F8      0604     60      LD      B, 4
A7FA      2178AB   70      LD      HL, 43896
A7FD      CD18BB   80      INLOOP  CALL  GETKEY
AB00      FE80     90      CP      #80
A802      2001     100     JR      NZ, NOT_0
A804      AF       110     XOR      A
A805      77       120     NOT_0    LD      (HL), A
A806      23       130     INC      HL
A807      10F4     140     DJNZ     INLOOP
A809      CDB4AA   150     CALL    43700
A80C      213CA8   160     LD      HL, D_MSG
AB0F      7E       170     MSG_LP   LD      A, (HL)
ABI0      CD5ABB   180     CALL    PRINT
A813      23       190     INC      HL
A814      FE00     200     CP      #00
A816      20F7     210     JR      NZ, MSG_LP
A818      217BAB   220     LD      HL, 43899
A81B      AF       230     XOR      A
A81C      CB3E     240     SRL      (HL)
A81E      0603     250     LD      B, 3
AB20      2B       260     DIV_LP   DEC      HL
A821      CB1E     270     RR      (HL)
A823      10FB     280     DJNZ     DIV_LP
AB25      F5       290     PUSH     AF
AB26      CDB4AA   300     CALL    43700
AB29      3E20     310     LD      A, 32
A82B      CD5ABB   320     CALL    PRINT
A82E      3E52     330     LD      A, "R"
A830      CD5ABB   340     CALL    PRINT
A833      F1       350     POP      AF
A834      CE00     360     ADC      A, 0
A836      F630     370     OR      #30
A838      CD5ABB   380     CALL    PRINT
A83B      C9       390     RET
A83C      20446976 400     D_MSG   DEFM   " Div "
A840      69646564 410     DEFM   "ided"
A844      20627920 420     DEFM   " by "
A848      74776F3D 430     DEFM   "two="
AB4C      2000     440     DEFW    #0020

```

Pass 2 errors: 00

Table used: 134 from 306

Executes: 43000

ura 11.12. Sumas 046C,0499,0486, 041F, 057D, 0565, 0503, 0390, 01B7.

pre los mismos. Cuando se realiza una rutina de división para números cualesquiera, es esencial asegurarse de que no se va a producir ningún intento de dividir por 0. Si no se toma esta precaución, una división por 0 nunca puede concluir, ya que el resultado es infinito.

Existen muchas formas de comprobar que el divisor no es 0. Para 8 bits, se puede cargar el divisor en A y efectuar un AND A. Para 16 bits se puede cargar un byte del divisor en A y efectuar un OR con el otro byte. Ambos métodos activarán el indicador de cero si el divisor es 0.

Ahora podemos realizar la división por 2 de un número del tamaño que queramos. Habrá que usar SRL o SRA (según que el número sea sin signo o con signo) en el byte más significativo, seguido de RR en cada uno de los siguientes bytes. Esto es lo que hemos hecho en el programa de la figura 11.12, preparado para utilizar también la rutina de impresión de 6.13.

Para permitirle que introduzca el dividendo, la rutina de entrada capta el código ASCII de la tecla que se pulse y lo interpreta como un byte de un número de 32 bits. Hay que pulsar, pues, 4 teclas. La primera se interpreta como el byte menos significativo y las siguientes van aumentando en significación. Existe un problema: el código ASCII 0 (NUL) no puede ser introducido desde el teclado del Amstrad; lo hemos solucionado haciendo que se obtenga el código 0 cuando usted pulse la tecla '0' del teclado numérico. Puede usted objetar que su Guía del usuario afirma que el código 0 se obtiene con [CONTROL]A; pero debe observar que, según la misma Guía, se obtienen dos códigos diferentes con [CONTROL]C. Lo que ocurre de hecho es que [CONTROL]A corresponde al código 1, [CONTROL]B al 2, [CONTROL]C al 3 y, a partir de ahí, todo sigue como dice la Guía.

El indicador de arrastre es fundamental, porque almacena los restos que se van produciendo. Pero no es necesario preservar los indicadores antes de ejecutar la instrucción DJNZ DIV\_LP, ya que no les afecta para nada. Sin embargo, al terminar la división, sí es necesario almacenar hasta después el acumulador A (cargado con 0 y listo para la instrucción ADC posterior) y los indicadores (el de arrastre, con el resto que se imprimirá al final).

Experimente con este programa hasta que esté seguro de comprender bien cómo se realiza la división mediante desplazamientos y rotaciones.

El programa de la figura 11.13 es el equivalente para la división del programa de la figura 11.8. También utiliza la rutina de impresión de 6.13.

El programa realiza, como el de 11.8, una pasada de bucle por cada cifra binaria del divisor. El dividendo se carga en el registro E y el divisor en el C; el registro B es el contador, y se lo actualiza con la instrucción DJNZ. En cada pasada del bucle, las rotaciones depositan en la posición menos significativa de E (el dividendo) el bit de arrastre anterior, mientras que el bit más significativo de E pasa al indicador de arrastre y de ahí a la posición menos significativa de A. Al principio, el indicador de arrastre está a 0 a



Hisoft GENA3 Assembler. Page 1.

## Pass 1 errors:

```

                                DIVISION USANDO DESPLAZAMIENTO
                                10 ; y ROTACION
A7F8                            20      ORG      43000
A7FB                            30      ENT      43000
BB1B                            40 GETKEY EQU      47896
BB5A                            50 PRINT EQU      47962
A7F8 210000                    60      LD      HL,0
A7FB 2278AB                    70      LD      (43896),HL
A7FE 227AAB                    80      LD      (43898),HL
A801 CD40AB                    90      CALI- GETVAL
A804 5F                        100     LO      E, A
A805 --2156A8                  110     LD      HL, D_MSG
Asea CD4CA8                    120     CALL MSG_LP
A80B CD40A8                    130     CALL GETVAL
A80E 4F                        140     LD      C, A
A80F 2164A8                    150     LD      HL,MSG2
A812 CD4CAB                    160     CALL MSG_LP
A815 AF                        170     XOR      A
A816 0608                      180     LD      B, 8
A818 C813                      190 DIV_LP RL      E
A81A 17                        200     RLA
A81B 91                        210     SUB      C
A81C 3001                      220     JR      NC, NO_ADD
A81E 81                        230     ADD      A, C
A81F 10F7                      240 NO_ADD DJNZ DIV_LP
A821 47                        250     LD      B, A
A822 7B                        260     LD      A, E
A823 17                        270     RLA
A824 2F                        280     CPL
A825 CD33A8                    290     CALL P_NUMB
AB28 216BA8                    300     LD      HL,MSG3
A82B CD4CA8                    310     CALL MSG_LP
A82E 78                        320     LD      A, B
AB2F CD33A8                    330     CALL P_NUMB
A832 C9                        340     RET
A833 E5                        350 P_NUMB PUSH HL
A834 D5                        360     PUSH DE
A835 C5                        370     PUSH BC
AB36 3278AB                    380     LD      (43896), A
AB39 CDB4AA                    390     CALL 43700
A83C C1                        400     POP      BC
A83D D1                        410     POP      DE
A83E E1                        420     POP      HL
A83F C9                        430     RET
AB40 CD18BB                    44C GETVAL CALL GETKEY
A843 F5                        450     PUSH AF
A844 CD33A8                    460     CALL P_NUMB
A847 F1                        470     POP      AF
A848 A7                        480     AND      A
A849 0E                        490     RET      NZ
AB4A E1                        500     POP      HL
A84B C9                        510     RET
A84C 7E                        520 MSG_LP LD      A, (HL)

```

AB4D	CD5ABB	536	CALL	PRINT
A850	23	546	INC	HL
A851	FE00	550	CP	#00
A853	20F7	560	JR	NZ,MSG_LP
A855	C9	570	RET	
A856	20446976	580	DEFM	" D i v "
A85A	69646561	590	DEFM	" i d e d "
AB5E	20627920	600	DEFM	" by "
AB62	2000	610	DEFW	#0020
A864	3D0D	620	MSG2	DEFW #0D3D
A866	0A00	630	DEFW	#000A
A86B	2052	640	MSG3	DEFW #3220
AB6A	2000	650	DEFW	#0020

Figura 11.13. Sumas: 037A, 04F4, 04D4, 0256, 0430, 0637, 06AC, 06B8, 0692, 03F0, 024E, 009C

consecuencia de la instrucción XOR A, que se emplea para poner A a 0. Tras las rotaciones, se hace un intento de restar el divisor de A. Si se produce arrastre, lo que significa que la resta es imposible, se restituye a A su valor original sumando el divisor.

Como ocurría con la multiplicación, se emplea también aquí el procedimiento de división que se aprende en la escuela para las divisiones largas. En la figura 11.14 se puede ver cómo es este proceso en el caso de la división de 85 por 2. Los bits de arrastre que salen o entran en los registros se indican con flechas.

( 1 )			01010101	00000000
DIV_LP	RL E		0<-10101010<-0	00000000
	RLA			0<-00000000<-0
	SUB C			1<-11111110
	JR NC,NO_ADD			
	ADD A,C			1<-00000000
NO_ADD	DJNZ DIV_LP			
( 2 )				
DIV_LP	RL E		1<-01010101<-1	0<-00000001<-1
	RLA			1<-11111101
	SUB C			1<-00000001
	JR NC,NO_ADD			
	ADD A,C			
NO_ADD	DJNZ DIV_LP			
( 3 )				
DIV_LP	RL E		0<-10101011<-1	0<-00000010<-0
	RLA			0<-00000000
	SUB C			
	JR NC,NO_ADD			
NO_ADD	DJNZ DIV_LP			
( 4 )				

```

DIV_LP  RL  E          1<-01010110<-0
        RLA          0<-00000001<-1
        SUB  C        1<-11111101
        JR   NC,NO_ADD
        ADD  A,C        1<-00000001
NO_ADD  DJNZ  DIV_LP
( 5 )
DIV_LP  RL  E          0<-10101101<-1I
        RLA          0<-00000010<-0
        SUB  C        0<-00000000
        JR   NC,NO_ADD
NO_ADD  DJNZ  DIV_LP
(6)
DIV_LP          RL      E  1<-01011010<-0
        RLA          0<-00000001<-1
        SUB          c    1<-11111101
        JR   NC,NO_ADD
        ADD  A,C        1<-00000001
NO_ADD  DJNZ  DIV_LP
( 7 )
DIV_LP  RL  E          0<-10110101<-1
        RLA          0<-00000010<-0
        SUB  C        0<-00000000
        JR   NC,NO_ADD
NO_ADD  DJNZ  DIV_LP
(8)
DIV_LP  RL  E          1<-01101010<-0
        RLA          0<-00000001<-1
        SUB  C        1<-11111101
        JR   NC,NO_ADD
        ADD  A,C        1<-00000001
NO_ADD  DJNZ  DIV_LP
        LD  B,A      & es ahora (0000000)
        LD  A,E
        RLA          01101010
        CPL          0<-11010100<-1
                   00101010

```

Figura 11.14

Conviene que vuelva una y otra vez sobre este proceso hasta que lo entienda perfectamente. Se conoce este método de división como método de restauración (*restoring*), puesto que restaura el contenido previo a la resta cuando se produce arrastre. Hay otros métodos de división, pero quedan fuera del propósito del libro. El que hemos visto es eficaz y se adapta fácilmente para números de más de un byte, por lo que permite realizar cualquier división.

Existe otra forma interesante de utilizar las rotaciones y los desplazamientos. Introduzca el programa de la figura 11.15 y, tras haberlo grabado en cinta, ponga el Amstrad en modo 2 (si se está utilizando el ensamblador, el comando W permite hacerlo) y llene la pantalla con muchos caracteres, por

ejemplo, listando el CARGADOR HEX o provocando mensajes de error.

Ejecute entonces el programa. Observará que el contenido de la pantalla se desplaza a la derecha un punto (*pixel*) cada vez, hasta un total de un carácter. Cambie la instrucción RR por alguna otra de las instrucciones vistas en este capítulo y trate de predecir el resultado. Note que el registro de estado se preserva cuidadosamente. ¿Qué ocurrirá si se suprimen las instrucciones PUSH y POP? Compruébelo. Ensaye también con los otros modos y verá cómo se producen efectos curiosos.

```

Hisoft GENA3 Assembler. Page      1.

Pass 1 errors: 00

                                10 ; FIG. 11.15 -DESPLAZAMIENTO A LA
                                20 DERECHA DE LA PANTALLA
A7F8                            30      ORG 43000
A7F8                            40      ENT 43000
A7F8 0608                       50      LD B,8
A7FA F5                         60      PUSH AF
A7FB 2100C0                     70 SCREEN LD HL,#C000
A7FE F1                         80 PIXEL POP AF
A7FF CB1E                       90      RR (HL)
A801 F5                        100     PUSH AF
A802 23                        110     INC HL
A803 7D                        120     LD A,L
AB04 B4                        130     OR H
A805 20F7                     140     JR NZ,PIXEL
A807 10F2                     150     DJNZ SCREEN
A809 F1                       160     POP AF
A80A C9                       170     RET

Pass 2 errors: 00

Table used;      38   from    143
Executes: 43000

```

Figura 11.15. Sumas de comprobación: 04B3, 0527.

La figura 11.16 proporciona el programa análogo para el desplazamiento a la izquierda. Observe los cambios que hemos realizado.

Con la ayuda del mapa de pantalla del apéndice F, aprenderá a realizar programas que desplacen solamente ciertos trozos de la pantalla. Los dos programas que hemos visto pueden parecer lentos, pero, si tiene en cuenta que cada desplazamiento en un punto (*pixel*) lleva 16384 rotaciones y casi 132000 instrucciones, apreciará la velocidad a la que se realiza.

Existen, aún otras dos instrucciones de rotación, que podrá ver si lo desea en el apéndice A. Se denominan *rotaciones decimales*. Quedan fuera del pro-

Hisoft GENA3 Assembler. Page 1

Pass 1 errors: 00

```

10  i DESPLAZAMIENTO A LA
20  IZQUIERDA DE LA PANTALLA
A7FB 30 ORG 43000
A7FB 40 ENT 43000
A7FB 0608 50 LD B,B
A7FA F5 60 PUSH AF
A7FB 21FFFF 70 SCREEN LD HL,#FFFF
A7FE F1 80 PIXEL POP AF
A7FF CB16 90 RL (HL)
A801 F5 100 PUSH AF
A802 2B 110 DEC HL
A803 7D 120 LD A,L
A804 A7 130 AND A
A805 20F7 140 JR NZ,PIXEL
A807 7C 150 LD A,H
A808 FEC0 160 CP #C0
A80A 20F2 170 JR NZ,PIXEL
A80C 10ED 180 DJNZ SCREEN
A80E F1 190 POP AF
A80F C9 200 RET

```

Pass 2 errors:

```

Table used: 38 from 141
Executes: 43000

```

Figura 1.1.16. Sumas de comprobación: 05E9, 05B2, 02B7.

pósito de este libro y lo normal es que no tenga necesidad de utilizarlas, salvo para algún trabajo de pantalla. Están pensadas para utilizar con números decimales codificados en binario, que se utilizan en sistemas antiguos como, por ejemplo, las pantallas de los relojes digitales. El sistema de codificación binaria de los números decimales (*Binary Coded Decima!* o *BCD*) utiliza un código de 4 bits para las cifras del 0 al 9. De esta manera sólo los números comprendidos entre 0 y 99 pueden ser codificados en un byte, mientras en la forma hexadecimal habitual se puede representar de 0 a 255. Si está interesado en estas instrucciones, lo mejor es que asimile primero los conceptos de este libro y luego pase a leer libros como el de R. Zaks 'Programming the Z80' SYBEX (ISBN 0 89588 069 5).

## Resumen

Vamos a resumir las instrucciones explicadas en este capítulo. Utilizaremos los símbolos:

- r = cualquiera de los registros de 8 bits (A, B, C, D, E, H o L)
- m = cualquiera de los r y (HL)
- rr = cualquier par de registros que se utilicen como uno de 16 bits
- n = un número de 8 bits, o sea, entre 0 y 255
- mn = un número de 16 bits, o sea, entre 0 y 65535
- ( ) rodeando un número o un par de registros = el contenido de la dirección.
- PC = contador de programa
- SP = puntero de pila

Los desplazamientos y rotaciones pueden usar cualquier m.

Existen códigos especiales de 1 byte para las rotaciones del registro A. Estas rotaciones especiales sólo afectan al indicador de paridad.

Todas las otras rotaciones y desplazamientos afectan a todos los indicadores, en el sentido que corresponda al valor almacenado en m tras la operación.

El indicador P/V tiene el sentido de indicador de paridad.

Las rotaciones decimales no afectan al indicador de arrastre.

En la división de un número con signo se debe conservar el signo utilizando la instrucción SRA.

Las rotaciones circulares no recogen el contenido que hubiera en el indicador de arrastre antes de la operación.

Los movimientos a la derecha dividen por 2.

Los movimientos a la izquierda multiplican por 2.

## Búsquedas y transferencias automáticas

Algunas instrucciones del Z80 ejercen, al ser ejecutadas, cierto control sobre su propio efecto; por ello vamos a denominarlas "instrucciones automáticas". Ya hemos visto una de ellas en capítulos anteriores; se trata de la instrucción DJNZ. Esta instrucción efectúa un salto condicionado a un NZ, pero al mismo tiempo se encarga de actualizar su contador, que es el registro B.

De las restantes instrucciones que poseen este carácter automático, unas sirven para realizar búsquedas o transferir datos; son las que estudiaremos en este capítulo. Las otras realizan las entradas y salidas de la información, que es un tema que abordaremos en el capítulo siguiente.

Supongamos que hay que almacenar el contenido de un área de la memoria en otro área. Hay muchas ocasiones en que esto es necesario; por ejemplo, para crear huecos en una serie de registros de una base de datos, para guardar el contenido de una pantalla o incluso para mover la pantalla de manera similar a como hemos hecho en las figuras 11.15 y 11.16, etc. Si el bloque de memoria que hay que mover es de tamaño conocido, el programa será más o menos como el de la figura 12.1.

La secuencia EX DE,HL LD (HL),A EX DE,HL es puramente gratuita y se puede sustituir por LD (DE),A que hace lo mismo; la hemos incluido para mostrar cómo se utilizan las instrucciones EX. Si no conviene utilizar el registro A y hay que transferir menos de 257 bytes, se puede reescribir el programa para que utilice el registro B y controle el bucle con la instrucción DJNZ.

Como contador se usa el par BC (que se puede recordar como *Binary Counter* o contador binario) y DE contiene la dirección de destino (DE recuerda Destino) del byte. HL desempeña su papel tradicional de puntero, en este caso de la dirección de origen.

El programa de 12.1 funcionará correctamente siempre que la dirección de destino del bloque sea inferior a la de origen del bloque. En caso contrario puede no dar el resultado apetecido. Por ejemplo, si el programa se completa haciendo

```
ORIGIN EQU #C000
DEST EQU #C100
COUNT EQU #3EFF
```

(con el CARGADOR HEX se completará convenientemente el código de las líneas 60, 70 y 80 y se utilizarán las sumas de comprobación 036F 04CC 00C9) el resultado será la repetición varias veces del mismo trozo de la pantalla, que no era lo que se pretendía. La transferencia se hace en la forma deseada para las primeras FFh posiciones, pero luego se repite constantemente este mismo trozo, ya que las posiciones de origen habrán sido alteradas antes de la transferencia.

```

Hisoft  GENA3  Assembler.  Page      1.

Pass  1  errors; 00

                                1  ; FIG. 12.1 -- TRANSFERENCIA DE BLOQUES
                                EN SENTIDO CRECIENTE

4E20                                10      ORG  20000
4E20                                20      ENT  20000
0000                                30  ORIGIN EQU  #????
0000                                40  DEST  EQU  #????
0000                                50  COUNT EQU  #????
4E20  210000                        60      LD  HL,ORIGIN
4E23  110000                        70      LD  DE,DEST
4E26  010000                        80      LD  BC,COUNT
4E29  7E                            90  LOOP  LD  A,(HL)
4E2A  EB                           100     EX  DE,HL
4E2B  77                           110     LD  (HL),A
4E2C  EB                           120     EX  DE,HL
4E2D  23                           130     INC HL
4E2E  13                           140     INC DE
4E2F  0B                           150     DEC BC
4E30  78                           160     LD  A,B
4E31  B1                           170     OR  C
4E32  20F5                         180     JR  NZ,LOOP
4E34  C9                           190     RET

Pass  2  errors: 00

Table  used:      60      from      147
Executes:  20000

```

Figura 12.1

Vamos a comentar esto con ayuda de un ejemplo elemental. Si para remediar el error de la frase

#### Hacer una transferencia

realizamos la lógica transferencia de letras hacia adelante, que deje sitio para intercalar la 'e', con origen nn+18, destino nn+19 y contador 4 (nn es la posición de H), lo que obtendremos tras cada una de las pasadas del bucle será



Hacer una transfernnia  
 Hacer una transfernnna  
 Hacer una transfernnnn  
 Hacer una transfernnnnn

lo que empeora la situación.

Antes de ver cómo se pueden solucionar estos problemas, vamos a introducir la primera instrucción de transferencia automática de bloques. Ella sola puede reemplazar todas las instrucciones que figuran entre las líneas 90 a 180, ambas inclusive, del programa precedente. Es la instrucción LDIR (de *LoaDing Incrementing Repeating*, o sea, carga incremento repetición) y su funcionamiento ha quedado explicado al decir qué instrucciones reemplaza en 12.1. Sus códigos son

ENSAMBLADOR	HEX	BINARIO
LDIR	ED B0 11 101 101 10 110 000	

```

Hisoft GENA3 Assembler. Page      1.

Pass 1 errors: 00

1 ; FIG. 12.2 - TRANSFERENCIA DE BLOQUES
  EN SENTIDO DECRECIENTE
4E20          te      ORG 20000
4E20          20      ENT 20000
FEFF          30     ORIGIN EQU #FEFF
FFFF          40     DEST  EQU #FFFF
3EFF          50     COUNT EQU #3EFF
4E20  21FFFE     60      LD  HL,ORIGIN
4E23  11FFFF     70      LD  DE,DEST
4E26  01FF3E     B0      LD  BC,COUNT
4E29  7E         90     LOOP LD  A,(HL)
4E2A  EB        100      EX  DE,HL
4E2B  77        110      LD  (HL),A
4E2C  EB        120      EX  DE,HL
4E2D  2B        130      DEC HL
4E2E  1B        140      DEC DE
4E2F  0B        150      DEC BC
4E30  78        160      LD  A,B
4E31  B1        170      OR  C
4E32  20F5      180      JR  NZ,LOOP
4E34  C9        190      RET

Pass 2 errors: 00

Table used: 60 from 147
Executes: 20000

```

Figura 12.2.

Cuando la dirección de destino del bloque es superior a la de origen, hay que sustituir el programa de 12.1 por el de la figura 12.2, que realiza la transferencia en orden inverso, o sea, empezando por la dirección más alta del bloque (tanto en origen como en destino).

La instrucción que reemplaza las que figuran en 12.2 entre las líneas 90 a 180 es ahora LDDR (de *LoadIng Decrementing Repeating*, o sea, carga disminución repetición). Sus códigos son

ENSAMBLADOR	HEX	BINARIO
LDDR	ED B8	11 101 101 10 111 000

Aunque las instrucciones LDIR y LDDR pueden servir para realizar las mismas funciones, no operan de la misma manera. LDIR desplaza un bloque comenzando por las direcciones bajas de origen y destino; necesita que HL y DE estén cargados al comienzo con las direcciones bajas del bloque de origen y el bloque de destino respectivamente. Conviene emplear LDIR cuando la dirección de destino es más baja que la de origen. LDDR desplaza un bloque comenzando por las direcciones altas de origen y destino; necesita que HL y DE estén cargados al comienzo con las direcciones altas del bloque de origen y el bloque de destino respectivamente. Conviene emplear LDDR cuando la dirección de destino es más alta que la de origen.

Bastantes cosas son comunes a LDIR y LDDR. No utilizan el registro A

```

Hisoft  GENA3 Assembler. Page      1.

Pass 1 errors: 00

                                1 ; FIG. 12.3 - LLENADO DE PANTALLA
4E20                                10      ORG  20000
4E20                                20      ENT  20000
FFFF                                30  ORIGIN EQU  #FFFF
FFFE                                40  DEST  EQU  <FFFE
3FFF                                50  COUNT EQU  #3FFF
4E20  21FFFF                        60      LD   HL,ORIGIN
4E23  11FEFF                        70      LD   DE,DEST
4E26  01FF3F                        90      LD   BC,COUNT
4E29  EDB8                          90      LDDR
4E2B  C9                           100      RET

Pass 2 errors: 00

Table used;      49  from      132
Executes: 20000

```

Figura 12.3. Sumas de comprobación: 0659, 0181.

en la transferencia. Utilizan el par BC como contador del número de bytes del bloque, y disminuyen su valor después de cada transferencia de un byte. Para comprobar si BC es cero (en cuyo caso se detiene la transferencia), no utilizan el indicador de cero, sino P/V; este indicador queda siempre a 0 al final de la instrucción. Estas instrucciones no afectan a los restantes indicadores accesibles.

Estas dos instrucciones pueden servir también para rellenar un área de memoria con un mismo byte; para ello se emplea de manera deliberada la técnica de 'sobrecopiado' desplazando el bloque 1 byte. Cada byte de destino se vuelve así byte de origen del siguiente traslado. El programa 12.3 da un ejemplo del empleo de esta técnica para rellenar la pantalla con un mismo carácter (el que hubiese en la posición FFFFh de memoria); en 9.2 hicimos algo parecido con otra técnica.

En los ejemplos anteriores se desplazaban bloques de longitud conocida, pero en muchas ocasiones lo que interesa es transferir bytes en tanto no se encuentre el límite deseado. El programa de la figura 12.4 realiza esta tarea

```

Hisoft GENA3 Assembler. Page      1.

Pass 1 errors: 00

                                1 ; TRANSFERIR HASTA ENCONTRAR UNO
4E20                          10      ORG  20000
4E20                          20      ENT  20000
FFFF                          30  ORIGIN EQU  #FFFF
FFFE                          40  DEST  EQU  #FFFE
3FFF                          50  COUNT EQU  #3FFF
4E20      21FFFF              60      LD   HL, ORIGIN
4E23      11FEFF              70      LD   DE, DEST
4E24      01FF3F              80      LD   BC, COUNT
4E29      7E                  90  LOOP  LD   A, (HL)
4E2A      12                  100     LD   (DE), A
4E2B      2B                  110     DEC  HL
4E2C      1B                  120     DEC  DE
4E2D      0B                  130     DEC  BC
4E2E      78                  140     LD   A, B
4E2F      B1                  150     OR   C
4E30      2804                160     JR   Z, LIMIT
4E32      AF                  170     XOR  A
4E33      BE                  180     CP   (HL)
4E34      20F3                190     JR   NZ, LOOP
4E36      C9                  200  LIMIT RET

Pass 2 errors: 00

Table used:      72      from  147
Executes: 20000

```

y el final se alcanza cuando se encuentra un byte 0. No obstante, se emplea BC para colocar un límite a la cantidad máxima de memoria que puede ser transferida. Si no se tomase esta precaución, podría suceder que la transferencia no terminase nunca; o, lo que es peor, que el programa escribiera encima de sí mismo, con la consecuencia que no es necesario describir.

Para esta finalidad, el Z80 dispone de dos instrucciones que son como LDDR y LDIR pero sin la repetición automática; son las instrucciones LDD y LDI, como era fácil imaginar. Sus códigos son

ASSEMBLER	HEX	BINARY
LDD	ED A8	11 101 101 1e 101 000
LDI	ED A0	11 101 101 10 100 000

No es tan simple como parece a primera vista modificar el programa de 12.4 para incluir la instrucción LDD, ya que ahora hay que utilizar el indicador P/V para detectar que el par BC ha llegado a 0, mientras que el programa original utilizaba el indicador de cero. La instrucción LDD pone a 0 el indicador P/V (o sea, en PO) cuando BC se hace 0. Como un salto relativo

```

Hisoft  GENA3 Assembler. Page      1.

Pass 1 errors: 00

1 ; TRANSFERIR HASTA ENCONTRAR UNO

4E20                10          ORG    20000
4E20                20          ENT    20000
FFFF               30 oRIGIN    EQU    #FFFF
FFFE               40 DEST      EQU    «FFFF
3FFF               50 COUNT     EQU    #3FFF
4E20 21FFFF        60          LD     HL,ORIGIN
4E23 11FEFF        70          LD     DE,DEST
4E26 01FF3F        80          LD     BC,COUNT
4E29 EDA8          90 LOOP      LDD
4E2B E2324E       171          JP     PO,LIMIT
4E2E AF           173          XOR    A
4E2F BE           174          CP     (HL)
4E30 20F7         180          JR     NZ,LOOP
4E32 C9           190 LIMIT     RET

Pass 2 errors: 00

Table used:      72    from    141
Executes: 20000

```

Figura 12.5

no se puede condicionar con P/V, hay que sustituir JR por JP. Tras los cambios, el programa queda según se muestra en la figura 12.5.

Las siguientes instrucciones automáticas, que son las últimas que veremos en este capítulo, son las de búsqueda en bloques. Sus códigos son semejantes a los de transferencia de bloques pero, como indica su nombre, lo que hacen es buscar un byte con determinado valor en un bloque de memoria. Para seguir el mismo proceso que antes, damos en la figura 12.6 un programa (sin emplear estas instrucciones) que busca un byte con el valor 65 (el código de 'A') a partir de la posición FFFFh y a través de 3FFFh bytes recorridos en sentido decreciente.

```

Hisoft GENA3 Assembler.  Page      1.

Pass 1 errors: 00

                                10      ;   BUSQUEDA DE UN BLOQUE
4E20                          20      ORG   20000
4E20                          30      ENT   20000
FFFF                          40  START EQU   #FFFF
3FFF                          50  COUNT EQU   #3FFF
4E20  21FFFF                  60      LD   HL, START
4E23  01FF3F                  70      LD   BC, COUNT
4E26  3E41                    80      LD   A, 65
4E28  BE                      90  LOOP  CP   (HL)
4E29  2B                      100     DEC  HL
4E2A  0B                      110     DEC  BC
4E2B  2807                    120     JR   Z, DONE
4E2D  57                      130     LD   D, A
4E2E  78                      140     LD   A, B
4E2F  B1                      158     OR   C
4E30  7A                      160     LD   A, D
4E31  20F5                    170     JR   NZ, LOOP
4E33  3F                      180     CCF
4E34  C9                      190  DONE  RET

Pass 2 errors: 00

Table used:    59   from    143
Executes: 20000

```

Figura 12.6

Al terminar el bucle (en la etiqueta DONE) quedará activado el indicador de cero si se ha encontrado el byte 65, y quedará desactivado si no se lo ha encontrado.

Mientras se realiza la comprobación de si BC es 0, es necesario almacenar el valor de A. Esto no se puede hacer con PUSH en la pila, pues entonces

se almacenaría F simultáneamente y, al recuperar A y F con POP, la comprobación posterior del indicador de cero sería inútil. Por eso se utiliza el registro D para almacenar el valor de A.

Hay una instrucción que realiza la misma tarea que este programa, o sea, *comparar, disminuir y repetir* la operación hasta que se termine el bloque previsto. Al acabar, el indicador de cero está a 1 si se ha encontrado el byte y está a 0 en caso contrario. La instrucción es CPDR y existe también la análoga CPIR (con incremento en lugar de disminución) y las correspondientes instrucciones sin repetición CPD y CPI. Sus códigos son:

ASSEMBLER	HEX	BINARY
<b>CPIR</b>	<b>ED B1</b>	<b>11 101 101 10 1 10 001</b>
<b>CPDR</b>	<b>ED B9</b>	<b>1 1 101 101 10 1 1 1 001</b>
<b>CPI</b>	<b>ED A1</b>	<b>1 1 101 101 10 100 001</b>
<b>CPD</b>	<b>ED A9</b>	<b>1 1 101 101 10 101 001</b>

En CPIR y CPDR el par HL marca el inicio del bloque en que se realiza la búsqueda, BC el tamaño del bloque y A contiene el valor del byte que se busca. CPIR incrementa HL en cada comparación; CDDR lo decrementa. La búsqueda concluye cuando se encuentra el byte o cuando BC se hace 0. Al terminar, el indicador de cero queda activado si se ha encontrado el byte.

Las instrucciones CPI y CPD funcionan de forma análoga, pero sin repetición.

Todas estas instrucciones utilizan el indicador P/V para indicar que BC se ha hecho 0, de la misma manera que las instrucciones de transferencia.

La figura 12.7 muestra el programa de 12.6 modificado para utilizar la instrucción CPDR.

El programa de la figura 12.7 puede a su vez ser transformado para realizar la búsqueda de una serie de bytes consecutivos, en lugar de un solo byte. Es frecuente usar esta técnica cuando se quiere encontrar una palabra o una frase entre un conjunto de datos almacenados en memoria, o para encontrar 'palabras clave' que se hayan preparado en un juego de aventuras. La figura 12.8 presenta un programa que sirve para buscar en la memoria una cadena literal que se le introduzca desde el teclado. Después de teclear los caracteres se debe pulsar [ENTER]; el programa entregará la dirección de la memoria en que comienza la cadena literal buscada, si ha podido encontrarla.

La rutina que permite la introducción de la cadena (líneas 120-190) se puede cambiar por otra, si conviene. Al terminar el programa, el par HL contiene la dirección donde comienza la cadena, si ha sido encontrada, o 0 en caso contrario. Para permitir comprobar el funcionamiento del programa, el con-

**Hisoft GENA3 Assembler. Page 1.**

**Pass 1 errors: 00**

```

                                10 ;   BUSQUEDA DE UN BLOQUE CON CPDR
4E20                          20      ORG   20000
4E20                          30      ENT   20000
FFFF                          40  START EQU   #FFFF
3FFF                          50  COUNT EQU   #3FFF
1E20  21FFFF                 60      LD    HL,START
4E23  01FF3F                 70      LD    BC,COUNT
4E26  3E41                   80      LD    A,65
4E28  EDB9                   90  LOOP   CPDR
4E2A  C9                    190  DONE   RET

```

**Pass 2 errors: 00**

**Table used: 59 from 132**

**Executes: 20000**

Figura 12.7. Sumas de comprobación: 0583, 00C9.

tenido final de HL se almacena también en la memoria. El siguiente programa BASIC le permitirá ejecutar el programa en código de máquina y comprobar los resultados; le sugerimos que ordene la búsqueda de la palabra 'HOLA' que figura en el programa BASIC.

```

10 PRINT "HOLA"
20 CALL 30000
30 N=PEEK(30069)+256*PEEK(30070):PRINT N
40 PRINT CHR$(PEEK(N));CHR*(PEEK(N+1));CHR
$(PEEK(N+2));CHR$(PEEK(N+3))

```

Conviene que reflexione un poco sobre el programa para asimilar enteramente su mecánica. La etiqueta FINÍ? no es necesaria, pero se la ha incluido para señalar el lugar en que el programa comprueba que ha encontrado completa la cadena que buscaba. A veces puede ocurrir que lo que el programa ha encontrado sea justamente la cadena introducida por el teclado, es decir, la propia muestra. Hay que tener cuidado de evitar esa posibilidad.

Como las instrucciones automáticas realizan cierta cantidad de operaciones simples, hay que tener claro el orden en que éstas se efectúan. Siempre modifican HL y, cuando lo utilizan, DE antes de disminuir BC. En consecuencia, la instrucción CPDR incrementa HL antes de disminuir BC; o sea, al final de la instrucción, HL apunta ya a la siguiente posición de memoria. Por eso al comienzo de NXT\_CH se incrementa DE pero no HL, que está

Hisoft 6ENA3 Assembler. Page 1.

Pass 1 errors: 00

```

10 ; FIG. 12.8 - BUSQUEDA DE UNACADENA
20 ; EN UN BLOQUE USANDO CPIR
7530          30      ORG      30000
7530          40      ENT      30000
BB18          50      GETKEY EQU 47996
BB5A          60      PRINT EQU 47962
0000          70      START EQU #0000
7530          B0      COUNT EQU 30000
7530 217575    90      LD      HL, FREE
7533 E5        100     PUSH   HL
7534 D1        110     POP     DE
7535 CD1BBB    120     INPUT   CALI- GETKEY
7538 77        130     LO      (HL), fi
7339 CD5ABB    140     CALL    PRINT
753C 23        150     INC     HL
753D FE0D      160     CP      #0D
733F 20F4      170     JR      NZ, INPUT
7541 210000    180     LD      HL, START
7544 013075    190     LD      BC, COUNT
7547 1A        200     LOOK    LD      A, (DE)
7548 D5        210     PUSH   DE
7549 EDB1      220     CPIR
754B C5        230     PUSH   BC
754C E5        240     PUSH   HL
754D 2012      250     JR      NZ, NOFIND
754F 13        260     NXT_CH  INC     DE
7550 1A        270     LD      A, (DE)
7551 BE        280     CP      (HL)
7552 23        270     INC     HL
7553 2BFA      300     JR      Z, NXT_CH
7555 FE0D      310     FINI?   CP      #0D
7557 E1        320     POP     HL
7558 C1        330     POP     BC
7559 D1        340     POP     DE
755fi 20E8     350     JR      NZ, LOOK
755C 2B        360     FOUND   DEC     HL
755D 227575    370     LD      (FREE), HL
7560 C?        380     RET
7561 C1        390     NOFIND  POP     BC
7562 E1        400     POP     HL
7563 D1        410     POP     DE
7564 23        420     INC     HL
7565 18F5      430     JR      FOUND

```

Pass 2 errors: 00

Table used: 145 from 184  
 Executes: 30000

Figura 12.8. Sumas de comprobación: 05A5, 0378, 04FD, 042E, 055E, 02E2.



ya apuntando a la posición siguiente. Por la misma razón parece la instrucción DEC HL en la etiqueta FOUND. Si no se ha encontrado la cadena buscada, la instrucción POP HL de la rutina NOFIND carga en HL el contenido de BC, que será 0 en ese caso; la instrucción INC HL compensa entonces la DEC HL que vendrá después.

Puede usted tratar de cambiar este programa para utilizar la instrucción CPDR en lugar de CPIR.

## Resumen

Vamos a resumir las instrucciones explicadas en este capítulo. Utilizaremos los símbolos:

- r = cualquiera de los registros de 8 bits (A, B, C, D, E, H o L)
- rr = cualquier par de registros que se utilicen como uno de 16 bits
- n = un número de 8 bits, o sea, entre 0 y 255
- nn = un número de 16 bits, o sea, entre 0 y 65535
- ( ) rodeando un número o un par de registros = el contenido de la dirección.
- PC = contador de programa
- SP = puntero de pila

LDIR carga el contenido de la dirección HL en la dirección DE, incrementa DE y HL, disminuye BC y, si BC no es 0, repite la operación (carga-incremento-repetición).

LDDR carga el contenido de la dirección HL en la dirección DE, disminuye DE y HL, disminuye BC y, si BC no es 0, repite la operación (carga-disminución-repetición).

LDI y LDD son como las anteriores, pero sin repetición.

CPIR compara el contenido de A con el contenido de la dirección HL, incrementa HL, disminuye BC y repite hasta que se produzca la igualdad o BC sea 0 (comparación-incremento-repetición). Si se ha producido la igualdad, e! indicador de cero queda activado.

CPDR compara el contenido de A con el contenido de la dirección HL, disminuye HL, disminuye BC y repite hasta que se produzca la igualdad o BC sea 0 (comparación-disminución-repetición). Si se ha producido la igualdad el indicador de cero queda activado.

CPI y CPD son como las anteriores, pero sin repetición.

En todas estas instrucciones el indicador P/V se pone a 0 cuando BC se hace 0; por lo tanto, si a continuación se hace JP PO, se efectuará el salto cuando BC sea 0.



## Comunicación con el exterior

Todas las instrucciones que hemos visto hasta ahora tenían como finalidad modificar y transportar información, pero sin salir del ordenador, o sea, limitándose a desplazamientos entre los registros y la memoria. Es posible que usted haya pensado a veces en cómo recoge el ordenador la información con la que trabaja; o tal vez se haya dicho que, cuando usted pulsa una tecla, el Amstrad se encargará de hacer lo que deba. De hecho, sí no necesitase información proveniente de fuera de ese mundo formado por la memoria, la pantalla y el microprocesador, el ordenador podría perfectamente olvidarse de usted y dedicarse a ejecutar sus programas, sin inmutarse aunque usted se dedicase a pulsar todas las teclas. La única cosa que podría usted hacer para perturbarle es desenchufar. Pero cuando el ordenador necesita información exterior, tiene los medios para conseguirla. El sistema operativo le proporciona la forma de acceder a lugares como el teclado o el generador de sonido, que no caen en el campo de acción directa del microprocesador. Sin entrar demasiado en detalles técnicos que nos harían salir del tema, vamos a dar una explicación elemental de la forma en que se comunica el microprocesador.

Hay dos cables perfectamente visibles que unen el ordenador y el monitor. Uno tiene dos hilos y sirve para suministrar electricidad al ordenador. El otro lleva dentro seis hilos, conectados a las seis clavijas del enchufe; por ese cable, el microprocesador envía información al monitor sobre la imagen que debe formar.

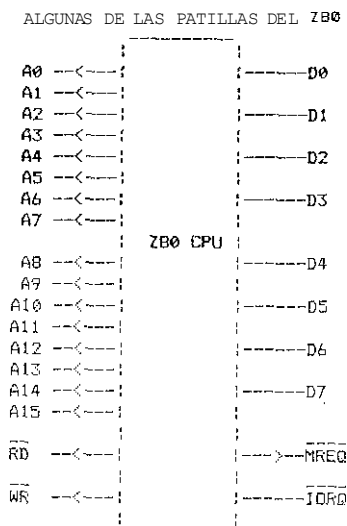
Pero lo que usted seguramente no ha visto es que el Z80 tiene 40 patillas, cada una con una misión específica. Hay 16 que se emplean para proporcionar la dirección con la que desea comunicarse el Z80. Otras 8 se utilizan para enviar o recibir los datos. Las restantes sirven para transmitir informaciones diversas como, por ejemplo, que se va a comunicar con la memoria, o con el exterior, o si la comunicación va a ser de entrada o de salida.

El conjunto de las 16 patillas que proporcionan las direcciones recibe el nombre de *bus de direcciones* (*address bus*). Sus patillas se representan mediante una A seguida del número que corresponde al bit que proporcionan. Van de la A0, que proporciona el bit 0 (o sea, el número 1 cuando está acti-

vada), a la A15, que proporciona el bit 15 (o sea, 32768 cuando está activada).

El conjunto de las 8 patillas que se emplean para la transmisión de datos se denomina justamente *bus de datos* (*Data Bus*). Las patillas llevan símbolos que van de D0 a D7, según el bit que representan.

En la figura 13.1 se dibujan esquemáticamente el bus de datos, el bus de direcciones y algunas de las patillas restantes.



RD indica petición de lectura (de *read*). WR indica petición de escritura (de *write*). MREQ indica que se va a utilizar la memoria (de *memory request*). JORQ indica petición de operación de entrada o de salida (de *input or output request*). La raya que se pone sobre estas patillas significa que están activadas cuando están a nivel bajo (binario 0).

Figura 13.1

Por ejemplo, cuando el Z80 ejecuta una instrucción tal como LD A,(3456), emite una señal para indicar que quiere usar la memoria y que quiere leer información en la dirección que ha colocado en el bus de direcciones. Entonces lee el contenido de dicha dirección de memoria a través del bus de datos.

Si se desea realizar una comunicación con otra cosa que no sea la memoria, habrá que indicarle al Z80 con qué debe comunicarse. Para ello están las instrucciones OUT (de *output*, salida) e IN (de *input*, entrada). El Z80

dispone de otras instrucciones de este tipo pero, debido a la forma en que está diseñado el Amstrad, sólo estas dos tienen interés. Sus códigos son:

ENSAMBLADOR		BINARIO						
OUT	(C), r	11	101	101	(EDh)	01	r	001
IN	r, (C)	11	101	101		01	r	000

La dirección con la que se debe establecer la comunicación de entrada o salida viene dada por el par BC. El registro B proporciona desde A8 a A15 y el registro C desde A0 hasta A7. Por ejemplo, cargando 1234h en BC, se tendrá 00010010 00110100.

Las direcciones de los elementos externos del equipo *no* reciben habitualmente este nombre. Se suele hablar *depuerta* (en inglés es *port*, puerto, pero en castellano se dice puerto o puerta, según los gustos). Así se evita cualquier confusión sobre si una dirección es interna (de memoria) o externa (de un dispositivo externo). Las operaciones realizadas a través de una puerta se llaman operaciones de E/S, o sea, entrada/salida (*I/O* en inglés).

Debido al diseño del Amstrad, hay pocos valores con los que se pueda cargar BC en este caso. Lo más probable es que usted utilice estas instrucciones para los distintos dispositivos periféricos. Los valores que quedan libres para B son F8h F9h FAh o FBh. Con todos ellos A10 está a 0 (a bajo nivel). Siempre que la línea de A10 esté a bajo nivel y que los bits A0... A7 estén carga-

```

Hisoft GENA3 Assembler. Page      1.

Pass 1 errors: 00

10 ; FIG. 13.2 - PROGRAMA PARA ENCENDER
20 ; Y APAGAR EL MOTOR DEL MAGNETOFONO
30 GETKEY EQU 47896
BB18
7530      40      ORG 30000
7530      50      ENT 30000
7530      60 ON   LD BC,#F6E0
7533      70      LD A, #10
7535      80      OUT (C), A
7537      90      CALL GETKEY
753A      AF      100     XOR A
753B      ED79     110     OUT (C), A
753D      C9      120     RET

Pass 2 errors: 00

Table used:      35 f
Executes: 30000

```

Figura 13.2. Sumas de comprobación: 052B, 02DE.

dos con valores entre E0h y FEh inclusive, no habrá posibilidad de interferencia con las direcciones reservadas por Amstrad para su uso actual o futuro en el CPC464. El Manual de referencia del programador le proporcionará detalles suplementarios sobre el equipamiento ligado al Amstrad.

El programa de la figura 13.2 muestra el uso de OUT para encender y apagar el motor del magnetófono del Amstrad. El magnetófono está al otro lado de un circuito de interfase (UPD 8255) que posee tres canales de E/S. El acceso para el canal A es la puerta F4xxh, para el canal B la puerta F5xxh y para el canal C la puerta F6xxh. El control se realiza por la puerta F7xxh. En todos los casos, xx puede ser cualquier valor (que será almacenado en el registro C) salvo uno de los no utilizados por A0...A7, que hemos mencionado antes, en cuyo caso tendría problemas.

## Otras instrucciones

En el Z80, los registros B, C, D de uso general, el acumulador A y el registro de estado F están duplicados y existen instrucciones para intercambiar valores entre los registros y los registros alternativos. Puede encontrar estas instrucciones en el Apéndice A, pero le aconsejamos vivamente que no las utilice, al menos sin conocer a fondo el sistema operativo del Amstrad. Olvide, pues, su existencia en tanto no domine completamente el Manual de referencia del programador.

La información que vamos a suministrarle en este capítulo le será muy útil si llega a asimilarla bien y a adquirir un buen conocimiento del sistema operativo. Ahora bien, es difícil precisar exactamente el grado de conocimiento que deberá poseer para sacar verdadero provecho de estas instrucciones.

### Interrupciones

El Amstrad genera interrupciones a intervalos regulares; es así como se las arregla para ejecutar instrucciones de EASIC tales como EVERY o AFTER. El Z80 puede reaccionar ante una interrupción de tres maneras diferentes, que son lo que se llama *modos de interrupción (interrupt modes)*; estos modos se representan por IM1, IM2 y IM3. Hay formas de seleccionar el modo de interrupción; el apéndice A proporciona las instrucciones necesarias. El programa de arranque en frío del Amstrad (recuerde que es el que se ejecuta cuando se enciende) selecciona para las interrupciones el modo 1 (IM1). Cuando se genera una interrupción en este modo, lo que se produce es una llamada a la dirección 56 (38h), en la que comienza un programa que suele recibir el nombre de *rutina del servicio de interrupciones*. Lógicamente, lo primero que hace esta rutina es almacenar el contenido de los registros, para poder devolver posteriormente los mismos valores cuando se vuelve al programa principal. Cuando una interrupción produce esta llamada, se detiene automáticamente cualquier otra interrupción que se esté llevando a cabo. Antes de volver al programa principal hay que desbloquear las interrupciones, para que las futuras interrupciones no sean ignoradas. La instrucción

que permite desbloquear las interrupciones es EI (de *Enable Interrupts*). Existe también la correspondiente instrucción que permite inhibir las interrupciones; es DI (de *Disable Interrupts*). Los códigos de estas instrucciones son

ENSAMBLADOR	HEX	BINARIO
DI	F3	11 110 011
EI	FB	11 111 011

Afortunadamente, Locomotive Software ha pensado en el programador de código de máquina y le ha proporcionado una manera sencilla de utilizar las interrupciones. En otras máquinas que utilizan el Z80, la gestión de interrupciones se realiza a través del modo 2, que es menos sencillo. Si usted desea utilizar su propia rutina de servicio para las interrupciones, lo que debe hacer es escribirla y añadir al final la instrucción JP #B939 en lugar de RET. A continuación debe ejecutar

```
LD HL,nn      (código hex  21 nn  )
LD (#39),HL    (código hex  22 39 00)
```

A partir de este momento, cada interrupción llamará a su rutina. Para desactivar su rutina de interrupción y volver a la situación normal ejecute

```
LD HL,#B939    (código hex  21 39 B9)
LD (#39),HL    (código hex  22 39 00)
```

No intente hacer el cambio de la dirección cargada en 39h en dos pasos separados, ya que, si ocurre una interrupción entre ambos, se llamará a una dirección equivocada.

Vamos a dar una breve descripción del modo IM2, aunque le recordamos que no debe usar este modo ni IM0 sin asimilar previamente lo que dice acerca de las interrupciones el Manual de referencia del programador. Con el IM2 se pueden utilizar las interrupciones para ejecutar las rutinas que se deseen, siempre que se desbloqueen las interrupciones antes de volver al programa y que se termine con la instrucción RETI. Debe recordar también que, antes de volver al BASIC, deberá restablecer el IM1 y desbloquear las interrupciones, salvo que se utilice RST 56 (38H) en la rutina de interrupción.

Al recibir una interrupción, el IM2 actúa de la manera siguiente: almacena el contenido del PC en la pila; inhibe las demás interrupciones; lee el valor 'bd' que haya en el bus de datos y el contenido del registro I (registro de interrupción); calcula la dirección bd+(256\*I); por fin, salta a la dirección que haya en dicha posición y la siguiente. Por ejemplo, si el registro I contiene



10 (0AH) y el dispositivo que realiza la interrupción coloca en el bus de datos el valor 200, entonces  $10 \times 256 = 2560$ , y  $2560 + 200 = 2760$ ; ahora, si la dirección 2760 contiene el valor 90 y la dirección 2761 contiene 187, la dirección de salto será  $90 + (256 \times 187)$  que es 47962. O bien, si 1 contiene 187 y el dispositivo envía el valor 90, entonces  $187 \times 256 = 47872$ , y  $47872 + 90 = 47962$ ; si 47962 contiene 207 y 47963 contiene 0, entonces  $0 + (207 \times 256) = 52992$  y el salto se efectúa a 52992.

Una manera sencilla de comprender lo que sucede es imaginarse que existe, justo en la posición anterior a la que se forma con I y con el valor del bus de datos, una instrucción invisible que dijese DI y CALL; de esta forma se saltaría a la dirección dada por las dos posiciones de memoria que vienen tras el CALL (dirección que se calcula en la forma habitual del Z80). Como la instrucción es invisible, no coloca la dirección de retorno en relación a sí misma; la que se almacena en la pila es la de la instrucción siguiente en el programa principal, que es a donde se volverá tras la instrucción RETI de la rutina de interrupción.

La instrucción RETI debe ir precedida de EI, como ya hemos dicho. La razón es que la llamada a la rutina de interrupciones lleva incorporado un DI para impedir que, como la rutina tardará en ejecutarse más tiempo del que media entre dos interrupciones, el programa caiga en un bucle sin fin.

Cualquier rutina de interrupción debe comenzar por preservar los valores de los registros en el momento de la entrada, para restablecer estos valores al volver al programa principal. No deben pasarse datos de la rutina por medio de los registros.

La utilización más típica de las interrupciones es el control de los movimientos en pantalla de figuras predefinidas (o *sprites*). La velocidad del movimiento de estas figuras se establece basándose en el conocimiento de la frecuencia con que se genera una interrupción. Como esto es independiente de cualquier otro aspecto del programa, se puede conseguir una velocidad constante de desplazamiento.

El programa de la figura 14.1 se compone de dos partes. La primera contiene dos rutinas: la primera modifica y la segunda restablece la dirección de la rutina del servicio de interrupciones. La segunda parte del programa es la rutina de interrupción alternativa; lo que hace es cargar 123 en la posición 31100. Antes de ejecutar ninguna parte de este programa, vuelva al BASIC y teclee

```
? PEEK (31100)
```

que le devolverá el valor 0,

```
POKE 31100,10: ? PEEK (31100)
```

que le devolverá 10 y

POKE 31100,0:PEEK(31100)

que le devolverá 0 otra vez. Ejecute ahora CALL 30000 y teclee de nuevo los mismos comandos. Ahora obtendrá siempre 123, ya que en cada interrupción se ejecuta la rutina final del programa. Ejecute CALL 30007 para volver a la rutina normal de interrupción y teclee otra vez los comandos. Todo habrá vuelto a la situación normal.

```

Hisoft  GENA3 Assembler.  Page      1.

Pass 1 errors: 00

                                1  ; FIG 11.1 - DESVIO DE LAS INTERRUPCIONES
7530                                10      ORG  30000
7530                                20      ENT  30000
7530  21 1879                      30  IN1T  LD   HL,31000
7533  223900                      40      LD   (#39),HL
7536  C9                          50      RET
7537  2139B9                      60  DISARM LD   HL,#B939
753A  223900                      70      LD   (#39),HL
753D  C9                          80      RET
7918                                90      ORG  31000
7919  F5                          100     PUSH AF
7919  3E7B                        110     LD   A,123
791B  327C79                      120     LD   (31100),A
791E  F1                          130     POP  AF
791F  C339B9                      140     JP   #B939

Pass 2 errors:

Table used:      37      from  142
Executes: 30000

```

Figura 14.1. Sumas de comprobación: 02E9, 0124 y 057B para la segunda parte.

Una última consideración sobre las interrupciones. Un programa en código de máquina irá más rápido si se inhiben las interrupciones con DI. Esto inhibirá también los comandos AFTER y EVERY de BASIC, pero no afectará prácticamente a nada más.

La siguiente instrucción nos será muy fácil de explicar después de lo anterior. Se trata de

ENSAMBLADOR	HEX	BINARIO
HALT	76	01 110 110

que es la instrucción cuyo código es el único de este tipo que no utilizaban las instrucciones LD r,r. La instrucción HALT detiene el Z80 en tanto no se reciba la siguiente interrupción. Si se ejecuta HALT cuando las interrupciones están inhibidas, provocará su detención total, por lo que hay que asegurarse de que las interrupciones están activadas cuando se las utiliza.

```

Hisoft  GENA3 Assembler.  Page      i.

Pass 1 errors: 00

                                10      FIG. 14.2 ■ RETARE
7530                                20      ORG   30000
7530                                30      ENT   30000
7530  FB                            40      EI
7531  06C8                          50      LD    B, 200
7533  76                            60  LOOP  HALT
7534  10FD                          70      DJNZ  LOOP
7536  C9                            80      RET

Pass 2 errors: 00

Table used:      24      ■ rom   136
Executes: 30000

```

Figura 14.2. Suma de comprobación: 0415.

Se usa normalmente esta instrucción en los programas de retardo, para conseguir grandes retrasos sin necesidad de recurrir a la utilización de bucles. La figura 14.2 muestra un programa en el que se utiliza HALT con esta intención. El retardo debido específicamente a la instrucción puede apreciarse probando también el programa con NOP en lugar de HALT; el cambio de instrucción puede hacerse tecleando POKE &7533,0.

## Reinicio (RST)

Existen ocho direcciones (todas ellas de la página 00h de memoria) que pueden ser llamadas mediante una instrucción de un sólo byte en lugar de la llamada habitual de tres bytes. Esta instrucción es conocida como *reinicio* {o *restart*) y su código es RST.

Las ocho direcciones posibles y los correspondientes códigos son:

ENSAMBLADOR	BINARIO	p	t	p	t
RST p	11 t 11 1	00h	000	20h	100
		08h	001l	28h	101
<b>RST 30h</b>	<b>11 110 111</b>	<b>10h</b>	<b>010</b>	<b>30h</b>	<b>110</b>
		18h	011	38h	111

Lo que hace esta instrucción es realizar una llamada a la dirección correspondiente, como si fuese una instrucción CALL; la rutina que empieza en esa dirección debe terminar por RET.

De las ocho direcciones posibles, la mayor parte son utilizadas por el Ams-

```

Hisoft GENA3 Assembler. Page      1.

Pass 1 errors: 00

10 ; F1G. 14.3 - DESVIO DE RST 30
20      ORG 30000
7530      ENT 30000
7530      3EC3      40      LD A, #C3
7532      215ABB      50      LD HL, #BB5A
7535      323000      60      LD (#30), A
7538      223100      70      LD (#31), HL
753B      3E48      B0      LD A, 72
753D      F7      90      RST #30
753E      3E65      100     LD A, 101
7540      F7      110     RST #30
7541      3E6C      120     LD A, 10B
7543      F7      130     RST #30
7544      3E6C      140     LD A, 108
7546      F7      150     RST #30
7547      3E6F      160     LD A, 111
7549      F7      170     RST #30
754A      C9      180     RET

Pass 2 errors: 00

Table used:      13  from      160
Executes: 30000

```

Figura 14.3. Sumas de comprobación: 02EC, 04B8, 040E.

trad para sus propias necesidades. Por ejemplo, 56 (38h) es la dirección de la rutina del servicio de interrupciones. Pero una de estas instrucciones, la RST 30h, está a disposición del programador.

El programa de arranque en frío prepara la dirección 30h de manera que se salte al programa de arranque al utiliza RST 30h; esto se puede comprobar tecleando CALL 48 (30h).

Para cambiar esta finalidad hay que colocar en la dirección 30h la instrucción de salto que convenga. Por ejemplo si usted utiliza con frecuencia la rutina PRINT de 47962 (BB5Ah) y decide que es mejor llamarla con RST 30h para ahorrar 2 bytes cada vez, lo que debe hacer es colocar en 30h el código de JP, y la correspondiente dirección en 31h y 32h (en la forma habitual). Esto es lo que hace el programa de la figura 14.3. En realidad, bastaría con lo que hay hasta la línea 50, colocando a continuación un RET; se ha añadido otra parte al programa para demostrar cómo funciona.

Ninguna de las instrucciones que hemos explicado hasta el momento en este capítulo afecta a los indicadores.

## Direccionamiento indexado

Hay dos registros de los que no hemos hablado todavía; son los IX y IY. Se llaman *registros índice* (de ahí la I), ya que se los utiliza para indicar direcciones de determinados elementos de información. Además, cada uno de ellos puede ser utilizado de la misma forma que el par HL.

Se preguntará usted cómo es posible que un sólo registro se pueda utilizar como un par. Pues bien, lo que ocurre es que tanto IX como IY son pares de registros que se utilizan al mismo tiempo como un registro de 16 bits. Los diseñadores del Z80 fueron incapaces de conseguir resultados fiables para estos pares cuando se los utilizaba separadamente. Por eso no publicaron las instrucciones que permiten utilizar cada registro de 8 bit de forma independiente.

En cuanto se conocen las instrucciones que utilizan IX y IY, es fácil descubrir cuáles son las que permiten usar separadamente los correspondientes registros de 8 bits. Todas las que hemos introducido en el Amstrad que se ha usado para el desarrollo de este libro han funcionado sin problemas. Claro que los ensambladores no recogen estas instrucciones, por lo que no pueden ser programadas con ellos. Además, no se puede garantizar que vayan a funcionar también en cualquier otro Amstrad CPC464, así que vamos a dejar las cosas como están y a realizar la descripción de las instrucciones que utilizan los registros índices IX y IY.

A excepción de las instrucciones ADC y SBC, todas las instrucciones que utilizan HL se pueden usar con IX y con IY. El código de una instrucción

que utilice IX es el mismo que el de la correspondiente instrucción para HL, pero debe llevar delante el byte DDh (221). Lo mismo ocurre cuando se emplea IY en lugar de HL; en este caso se añade el byte FDh (253). Por otra parte, cuando una instrucción utiliza HL en la forma (HL), la instrucción correspondiente con IX o IY lleva delante el byte suplementario, y lleva detrás otro byte que señala un desplazamiento en la forma de un número con signo; la instrucción afecta entonces a la posición apuntada por IX+d o IY+d, donde d es el desplazamiento. Todo esto resultará de momento un poco confuso, pero vamos a aclararlo con ejemplos.

El código para LD HL,nn es 21h seguido de los dos bytes que especifican el número nn de 16 bits. Cuando se utiliza IX hay que añadir el prefijo DDh, luego la instrucción resultará

```
ENSAMBLADOR  HEX
LD IX,nn      DD 21 nn
```

Cuando se utiliza IY la instrucción resulta

```
ENSAMBLADOR  HEX
LD IY,nn      FD 21 nn
```

Y de la misma forma todas las instrucciones que actúen directamente sobre los registros índice. Otros ejemplos son

ENSAMBLADOR	HEX	CON IX	CON IY
LD (nn),HL	22 nn	DD 22 nn	FD 22 nn
PUSH HL	E5	DD E5	FD E5
DEC HL	2B	DD 2B	FD 2B
JP (HL)	E9	DD E9	FD E9
ADD HL,B C	09	DD 09	FD 09

La última de las instrucciones sugiere una pregunta interesante. ¿Qué ocurre cuando en la instrucción ADD HL,HL se sustituye alguno de los HL por IX o IY? (Recuerde que ésta era una instrucción empleada para producir un desplazamiento a la izquierda de 16 bits en los programas de multiplicación.) Lo que sucede es que no se puede sustituir uno sólo de los HL, sino los dos simultáneamente. Si la instrucción ADD HL,HL va precedida de DDh, se convierte en ADD IX,IX; si va precedida de FDh, se convierte en ADD IY,IY.

Vamos a explicar ahora la transformación de las instrucciones que utilizan HL como puntero de una dirección en instrucciones que utilicen IX+d o IY+d con el mismo propósito. Es lo que se llama *direccionamiento indexado*. Será útil plantearse un caso práctico.

Supongamos que queremos tener almacenada la clásica agenda con direcciones y teléfonos. Uno de los principales problemas que suscita el almacenamiento y utilización de este tipo de datos es el de la diferente longitud de un mismo campo en los diferentes registros. Hay nombres más largos que otros, direcciones que ocupan diferente número de líneas, etc. Este tipo de problemas admite soluciones de dos tipos:

- 1) Reservar a cada campo la longitud que corresponda a la más larga de las que se van a necesitar.
- 2) Mantener en cada registro un índice con la longitud de cada línea, el número de líneas y la longitud total del registro.

El primer método es cómodo, pero antieconómico. El segundo parece muy difícil de realizar. Veamos que no es difícil con los registros índices.

Vamos a ver cómo se organizaría un registro que comenzase, por ejemplo, en la dirección 10000:

DIRECCIÓN	CONTENIDO
10000	byte bajo y
10001	byte alto de la longitud del registro, en 16 bits
10002	longitud del nombre
10003	longitud línea 1 de la dirección
10004	longitud línea 2 de la dirección
10005	longitud línea 3 de la dirección
10006	longitud línea 4 de la dirección
10007	longitud línea 5 de la dirección
10008	longitud del número de teléfono

Si el contenido del registro fuese

Jose Martínez López  
 Viriato 52  
 28010 MADRID  
 91 4458919

los contenidos de dichas direcciones serían

10000=51	longitud,byte bajo	10005 = 0	dirección 3
10001=0	longitud,byte alto	10006 = 0	dirección 4
10002=19	nombre	10007 = 0	dirección 5
10003 = 10	dirección 1	10008 = 10	teléfono
10004=12	dirección 2		

El índice ocupa 9 bytes (esto es lo mismo para todos los registros) y el registro ocupa 51. Como  $9 + 51 = 60$ , el índice del registro siguiente comenzará en 10060. Vamos a ver cómo se utilizan ahora los índices.

Si IX está cargado con 10000, entonces (IX+0) y (IX+1) darán la longitud total del registro, (IX+2) la longitud del nombre, y así sucesivamente. El comienzo del índice del siguiente registro se obtendrá siempre sumando 9, (IX+0) y  $256 \cdot (\text{IX}+1)$  a IX. Si se hace un programa que sirva para cargar un registro, elaborar su índice y pasar al registro siguiente, el programa servirá exactamente igual para cualquiera de los registros.

Las instrucciones que utilizan registros índice con desplazamiento tienen códigos nemotécnicos que resultan totalmente lógicos a estas alturas. Así, a LD A,(HL) le corresponden LD A,(IX+d) y LD A,(Y+d).

El desplazamiento es un número de 8 bits con signo, luego varía entre  $-128$  y  $+127$ ; ocupa 1 byte en los códigos y es obligatorio en las instrucciones que utilizan el registro para apuntar a una dirección de memoria, incluso aunque el desplazamiento sea 0. El código del desplazamiento va inmediatamente después del primer byte del código original. Por ejemplo,

LD A,(HL)	es 7Eh	LD A	(IX+d)	es DDh	7Eh d
INC(HL)	es 34h		INC(IX+d)	es FDh	34h d
RLC(HL)	es CBh	06h	RLC(IX+d)	es DDh	CBh d 06h
SET 4,(HL)	es CBh	E6h	SET 4,(IX+d)	es FDh	CBh d E6h
LD (HL),n	es 36h	n	LD (IX+d),n	es DDh	36 d n

Otra de las posibles utilizaciones de los registros índice consiste en realizar un cambio de los ejes de la pantalla, de manera que se pueda volcar el contenido de ésta a la impresora. La impresora recibe en cada byte información sobre una serie de puntos situados en vertical, mientras que para la pantalla un byte contiene información sobre puntos situados en horizontal. En modo 2, cada byte almacena la información de 8 puntos consecutivos; otro tanto ocurre con una impresora Epson, pero la dirección de la línea que forman los puntos es justamente perpendicular a la anterior. Por lo tanto, hay que hacer una rotación de 90 grados a la pantalla antes de copiarla directamente a la impresora.

Lo más económico no es rotar toda la pantalla al mismo tiempo, sino hacerlo con una línea de caracteres, mandarla a la impresora y hacer lo mismo con la línea siguiente y las demás. El programa de la figura J4.4 hace esto para la línea de pantalla de modo 2 que comienza en C000h. Como no todo el mundo tiene impresora, y no todas las impresoras utilizan los mismos códigos de control para ponerse en modo gráfico (si es que pueden hacerlo), hemos hecho que el programa actúe en la pantalla.

El mapa de pantalla cambia si se desplaza la pantalla hacia arriba. Para estar seguro de que comienza en C000h, utilice el comando MODE2 (o pulse



Hisoft GENA3 Assembler. Page 1.

Pass 1 errors: 00

```

1 i FlG. 14.4 - ROTACION DE PANTALLA
2 ; EN MODO 2
9C40          10          ORG      40000
9C40          20          ENT      40000
9C40 DD2100C0  30          LD       IX,#C000
9C44 2150C0    40          LD       HL,#C050
9C47 110000    50          LD       DE,#0800
9C4A 0650      60          LD       B, 80
9C4C DDE5      70 L0DP3    PUSH    IX
9C4E E5        80          PUSH    HL
9C4F C5        90          PUSH    BC
9C50 0608     100          LD       B,8
9C52 C5       110 LG0P2    PUSH    BC
9C53 E5       120          PUSH    HL
9C54 DDE5     130          PUSH    IX
9C56 0608     140          LD       B,8
9C5B DDCB0006 150 LOOP     RLC      ( IX+00)
9C5C CB1E     160          RR       (HL)
9C5E 19       170          ADD     HL, DE
<?C5F 10F7    180          DJN7    LOOP
9C61 DDE1     190          POP      IX
VC63 E1       200          POP      HL
9C64 DD19     210          ADD     IX, DE
9C66 C1       220          POP      BC
9C67 10E9     230          DJNZ    LOOP2
9C69 C1       240          POP      BC
9C6f1 E1      250          POP      HL
9C6B DDE1     260          POP      IX
9C6D DD23     270          INC      IX
9C6F 23       280          INC      HL
9C70 10DA     290          DJNZ    LOOP3
9C72 C9       300          RET

```

Pass 2 errors: 00

Table used: 48 from 147

Figura 14.4. Sumas de comprobación: 030B, 057A, 0467, 0586, 0656, 00C9.

W hasta conseguirlo, si está utilizando el ensamblador). Mueva el cursor a la primera línea y escriba en ella unos cuantos caracteres; pulse luego [ENTER], sin preocuparse porque dé un mensaje de error. Ejecute ahora el programa. La primera línea de caracteres aparecerá copiada en la segunda línea, pero cada carácter aparecerá girado 90 grados en el sentido de giro de las agujas del reloj.

Podrá usted adaptar este programa a sus necesidades, si es que está escri-

hiendo un programa para vuelco de pantalla o algo parecido. Hacer lo mismo en modos 0 y 1 es sensiblemente más difícil, a causa de la manera más complicada en que se almacena la información sobre cada punto.

Con esto termina el libro propiamente dicho y ahora lo que deberá hacer es practicar con lo que ha aprendido. En el siguiente capítulo le damos una idea sobre la utilización de las rutinas del sistema operativo y también algunos consejos finales. Lo que más le ayudará para las consultas que deba realizar serán los apéndices. Si se ha tomado en serio la programación, no le vendrá mal una plantilla para realizar diagramas de flujo. También podrá serle bastante útil alguna calculadora que trabaje en hexadecimal y en binario además de en decimal.

## Consejos sobre cómo utilizar el sistema operativo

El sistema operativo del Amstrad utiliza una serie de rutinas que se encargan del control de la pantalla, el magnetófono, el generador de sonidos, el manejo del teclado, etc. Están agrupadas en áreas que engloban todas las que se encargan de una misma tarea genérica.

No corresponde al propósito de este libro entrar en detalles sobre la configuración de estas secciones, ni realizar una descripción del sistema operativo; el Manual de referencia del programador cubre estos temas con gran detalle. Sin embargo, vendrán bien algunas consideraciones para permitirle iniciarse en el tema.

La forma en que el programador puede acceder al sistema operativo es realizar llamadas a ciertas direcciones de la memoria ROM. Estas direcciones están almacenadas en la memoria RAM, agrupadas en bloques de direcciones conocidos como *bloques de salto* (*jump-blocks*). La figura 15.1 muestra cuál es la técnica usual (y conveniente) para saltar a una de estas direcciones.

La utilización de los bloques de salto para el acceso a las rutinas presenta varias ventajas. Se puede cambiar la rutina a la que se llama de manera muy sencilla, con sólo cambiar la correspondiente dirección en el bloque, y sin necesidad de corregir el programa. También permite que el programa pueda calcular sus propias acciones.

Un caso típico en que este sistema es ventajoso se presenta cuando el programa realiza una serie de salidas por impresora. Se puede utilizar la pantalla mientras se depura el programa y, posteriormente, desviar la salida a la impresora. El coste del cambio consistiría sólo en alterar una dirección del bloque de salto. En BASIC se hace esto con mucha frecuencia, pero allí es todavía más sencillo ya que el dispositivo de salida que se asocia a una sentencia PRINT puede darse mediante una variable. En código de máquina no se emplean variables, pero se puede proceder como hemos explicado.

El propio Amstrad no puede utilizar este sistema a causa de los problemas de conmutación de áreas de memoria entre la RAM y la ROM. No se puede garantizar el acceso a la rutina correcta a menos que el programador cora-

## PROGRAMA PRINCIPAL

```
LD  A,ROUTNO ; numero (0...255) de la
                      rutina a la que se llama
CALL JUMP
```

## RESTO DEL PROGRAMA PRINCIPAL

```
JUMP:  ADD  A,A
        LD   D,0
        LD   E, A
        LD   HL,JBSTRT
        ADD  HL,DE ; HL contiene! ahora la
                      direccion de la memoria
                      que contiene la
                      dirección de la rutina

        LD   E,(HL)
        INC  HL
        LD   D,(HL)
        EX   DE,HL
        JP   (HL) ; salto a la rutina cuyo
                      RET final vuelve al
                      programa principal

JBSTRT: ; aqui empieza la lista de pares de bytes
          que almacenan las direcciones de las
          rutinas en la 'forma habitual'
```

Figura 15.1

pruebe que se encuentra activada el área de ROM que interesa, o la active en caso contrario. Además, si se necesita leer la pantalla, hay que desactivar la parte superior de la ROM, que se superpone a la pantalla.

La solución de Amstrad es reservar una cierta cantidad de instrucciones RST para realizar llamadas a las rutinas de la ROM. Esto garantiza que será activada la ROM que se necesite, y que la pantalla estará disponible si se requiere. Se extrae de la pila la dirección de vuelta de la instrucción RST y se la utiliza para examinar los bytes que siguen a la instrucción; en estos bytes se encuentra la dirección de la rutina a la que se llama y así se puede activar la ROM correspondiente. En el Manual de referencia del programador puede encontrar más en detalle cómo funcionan estas RST, pero es poco probable que esto le sea necesario por el momento.

Existen dos bloques de salto diferentes. Uno es el que utiliza el intérprete

de BASIC. El otro es el que constituye el sistema operativo o *firmware*. Se puede cambiar el que está activado en ese momento mediante un cambio de tres bytes, comenzando con la dirección RST. Si la nueva rutina está en el firmware, basta con copiar en la tabla de saltos los tres bytes correspondientes a la nueva rutina, en sustitución de los de la antigua. Si la nueva rutina es una desarrollada por usted, reemplácelos por un salto JP a la dirección de la rutina y, siempre que la rutina termine con un RET, todo funcionará normalmente. Mientras se cambie el bloque de salto principal, que se encuentra entre las direcciones BB00h y BDCBh, no se afecta para nada al funcionamiento de ninguna de las rutinas del sistema operativo. Sin embargo, si se altera el contenido de los bloques fuera de este área, se pueden producir efectos inesperados. Esto se debe a que ciertas rutinas pueden utilizar estos otros bloques de salto para realizar sus propias llamadas a rutinas que necesitan para su trabajo.

Si se encuentra en un apuro del que no sabe cómo salir, hay una rutina que es fundamental; es la de la dirección BD37h, que restablece los bloques de salto a su contenido original.

La pantalla y el sonido son tratados por hardware más que por software, lo que convierte en casi imposible el acceso directo a las funciones que desee utilizar, y lo mismo ocurre con el barrido del teclado. Afortunadamente, el firmware proporciona bastantes facilidades que le permitirán realizar esas tareas.

Los mandos de juego (*joysticks*) se pueden leer combinando dirección y botón de disparo, lo que permite detectar ocho direcciones de movimiento.

El apéndice G le proporciona las direcciones de llamada más usuales.

Cuando programe, utilice etiquetas alusivas al contenido de la sección que encabeza. Utilice también los comentarios, a pesar de que suponen un gasto de espacio; lo agradecerá el día en que deba volver a reparar el programa fuente y no recuerde ya la forma en que estaba organizado.

El libro no ha entrado en ciertos aspectos más profundos del Z80, ni se ha realizado ninguna consideración sobre los tiempos de ejecución. Su estudio le habrá permitido simplemente iniciarse en la programación y comenzar a utilizar el firmware. Para entrar en cosas más delicadas, o en lo que toca al hardware, podemos recomendarle algunos libros. Lea

Rodney Zaks "Programming the Z80" de SYBEX

para lo tocante a la programación. Para lo que se refiere al hardware o al desarrollo del Z80, puede interesarle

"The Z80 Tecnical Manual", de la propia casa ZILOG

o también

James Coffron "Z80 Applications" de SYBEX.

Cuando ya domine la programación en ensamblador, quizá le interese aprender otros lenguajes. Pascal es posiblemente el que le convendría aprender con vistas a programar en serio. Es parecido a BASIC en algunos aspectos, pero ofrece muchas de las posibilidades del ensamblador. Es un lenguaje compilado, como el ensamblador, pero tiene la ventaja de que un programa fuente es válido casi sin modificaciones, en máquinas diferentes de la que lo ha escrito. Existen muchos ordenadores para los que hay implementaciones de este lenguaje. Las principales limitaciones son la velocidad y el espacio; en ambos aspectos es peor que el ensamblador. Pero el Pascal es mucho más rápido que el BASIC y el código compilado ocupa mucho menos espacio que un programa BASIC comparable. Muchos programadores utilizan una mezcla de código generado por Pascal y, para las partes donde se pueden obtener mejoras importantes de espacio y velocidad, verdadero código de máquina obtenido a partir de ensamblador. Tanto Pascal como el ensamblador, una vez compilados, permiten separar el programa fuente del código objeto. El primero se almacenará para utilización posterior, si es necesaria. El código objeto se grabará y se cargará siempre que se desee ejecutar el programa.

Para el Amstrad existe una implementación de Pascal que suministra la casa Highsoft.

## Apéndice A

### Conjunto de instrucciones del Z80

El microprocesador Z80 tiene uno de los conjuntos de instrucciones más potentes y versátiles de los que existen en microprocesadores de 8 bits. Algunas instrucciones, que no posee ningún otro, permiten realizar cambios rápidos de bloques de memoria y transferencias entre la memoria y el exterior, de manera sencilla y eficaz.

A continuación damos un resumen del conjunto de las instrucciones del Z80 en el que se incluye el código nemotécnico de cada una, la operación que realiza, su interacción con los indicadores y algunos comentarios. Para más detalles se pueden consultar los manuales 'Z80 CPU Technical Manual' (03-0029-01) y 'Assembly Language Programming Manual' (03-0002-01).

Se han dividido las instrucciones en los siguientes grupos:

- carga de 8 bits
- carga de 16 bits
- intercambios, transferencia de bloques y búsqueda en bloques.
- operaciones aritméticas y lógicas de 8 bits.
- operaciones aritméticas y de control de aplicación general.
- operaciones aritméticas de 16 bits.
- rotaciones y desplazamientos.
- operaciones que trabajan con 1 bit.
- saltos
- instrucciones de llamada vuelta y reinicio.
- operaciones de entrada y salida.

## Grupo de carga de 8 bits

Código mnemotécnico	Operación simbólica	Indicadores					Códigos				N.º de bytes	N.º de ciclos M	N.º de estados T	Comentarios		
		S	Z	H	P/V	N	C	76	543	210					Hex	
LD r,r'	$r \leftarrow r'$	*	*	X	*	X	*	01	r	r'		1	1	4	r,r' Reg.	
LD r,n	$r \leftarrow n$	*	*	X	*	X	*	00	r	n10		2	2	7		
								$\leftarrow n \rightarrow$							000 B	
LD r,(HL)	$r \leftarrow (HL)$	*	*	X	*	X	*	01	r	110		1	2	7	001 C	
LD r,(IX+d)	$r \leftarrow (IX+d)$	*	*	X	*	X	*	11	011	101	DD	3	5	19	010 D	
								01	r	101					011 E	
								$\leftarrow d \rightarrow$							100 H	
LD r,(Y+d)	$r \leftarrow (Y+d)$	*	*	X	*	X	*	11	111	101	FD	3	5	19	101 I	
								01	r	110					111 A	
								$\leftarrow d \rightarrow$								
LD (HL),r	$(HL) \leftarrow r$	*	*	X	*	X	*	01	110	r		1	2	7		
LD (IX+d),r	$(IX+d) \leftarrow r$	*	*	X	*	X	*	11	011	101	DD	3	5	19		
								01	110	r						
								$\leftarrow d \rightarrow$								
LD (Y+d),r	$(Y+d) \leftarrow r$	*	*	X	*	X	*	11	111	101	FD	3	5	19		
								01	110	r						
								$\leftarrow d \rightarrow$								
LD (HL),n	$(HL) \leftarrow n$	*	*	X	*	X	*	00	110	110	36	2	3	10		
								$\leftarrow n \rightarrow$								
LD (IX+d),n	$(IX+d) \leftarrow n$	*	*	X	*	X	*	11	011	101	DD	4	5	19		
								00	110	110	36					
								$\leftarrow d \rightarrow$								
								$\leftarrow n \rightarrow$								
LD (Y+d),n	$(Y+d) \leftarrow n$	*	*	X	*	X	*	11	111	101	FD	4	5	19		
								00	110	110	36					
								$\leftarrow d \rightarrow$								
								$\leftarrow n \rightarrow$								
LD A,(BC)	$A \leftarrow (BC)$	*	*	X	*	X	*	00	001	010	0A	1	2	7		
LD A,(DE)	$A \leftarrow (DE)$	*	*	X	*	X	*	00	011	010	1A	1	2	7		
LD A,(nn)	$A \leftarrow (nn)$	*	*	X	*	X	*	00	111	010	3A	3	4	13		
								$\leftarrow n \rightarrow$								
								$\leftarrow n \rightarrow$								
LD (BC),A	$(BC) \leftarrow A$	*	*	X	*	X	*	00	000	010	02	1	2	7		
LD (DE),A	$(DE) \leftarrow A$	*	*	X	*	X	*	00	010	010	12	1	2	7		
LD (nn),A	$(nn) \leftarrow A$	*	*	X	*	X	*	00	110	010	32	3	4	13		
								$\leftarrow d \rightarrow$								
								$\leftarrow n \rightarrow$								
LD A,I	$A \leftarrow I$	1	1	X	0	X	IFF	0	11	101	101	ED	2	2	9	
								01	010	111	5F					
LD A,R	$A \leftarrow R$	1	1	X	0	X	IFF	0	11	101	101	ED	2	2	9	
								01	011	111	5F					
LD J,A	$J \leftarrow A$	*	*	X	*	X	*	11	101	101	ED	2	2	9		
								01	000	111	4F					
LD R,A	$R \leftarrow A$	*	*	X	*	X	*	11	101	101	ED	2	2	9		
								01	001	111	4F					

NOTAS: r,r' representan cualquiera de los registros A, B, C, D, E, H, L.  
 [FF] significa que el contenido ([FF] de la búscula de habilitación

de interrupciones se carga en P/V.  
 Para los diferentes símbolos que se emplean, véase la tabla que figura al final del apéndice.



## Grupo de carga de 16 bits

Código mnemotécnico	Operación simbólica	Indicadores S Z H P/V N C	Códigos 76 543 210 hex	N.º de bytes	N.º de ciclos M	N.º de estados T	Comentarios	
LD dd,nn	dd ← nn	* * X * X * * *	00 dd 001	3	3	10	dd	Par
LD IX,nn	IX ← nn	* * X * X * * *	11 011 101 DD 00 100 001 21	4	4	14	00 BC 01 DE 10 HL 11 SP	
LD IY,nn	IY ← nn	* * X * X * * *	11 111 101 FD 00 100 001 21	4	4	14		
LD HL,(nn)	H ← (nn + 1) L ← (nn)	* * X * X * * *	00 101 010 2A	3	5	16		
LD dd,(nn)	ddH ← (nn + 1) ddL ← (nn)	* * X * X * * *	11 101 101 ED 01 dd1 011	4	6	20		
LD IX,(nn)	IXH ← (nn + 1) IXL ← (nn)	* * X * X * * *	11 011 101 DD 00 101 010 2A	4	6	20		
LD IY,(nn)	IYH ← (nn + 1) IYL ← (nn)	* * X * X * * *	11 111 101 FD 00 101 010 2A	4	6	20		
LD (nn),HL	(nn + 1) ← H (nn) ← L	* * X * X * * *	00 100 010 22	3	5	16		
LD (nn),(dd)	(nn + 1) ← ddH (nn) ← ddL	* * X * X * * *	11 101 101 ED 01 dd0 011	4	6	20		
LD (nn),IX	(nn + 1) ← IXH (nn) ← IXL	* * X * X * * *	11 011 101 DD 00 100 010 22	4	6	20		
LD (nn),IY	(nn + 1) ← IYH (nn) ← IYL	* * X * X * * *	11 111 101 FD 00 100 010 22	4	6	20		
LD SP,HL	SP ← HL	* * X * X * * *	11 111 001 F9	1	1	6		
LD SP,IX	SP ← IX	* * X * X * * *	11 011 101 DD	2	2	10		
LD SP,IY	SP ← IY	* * X * X * * *	11 111 001 F9	2	2	10		
PUSH qq	(SP - 2) ← qqH (SP - 1) ← qqL SP ← SP - 2	* * X * X * * *	11 qq0 101	1	3	11	qq	Par
PUSH IX	(SP - 2) ← IXH (SP - 1) ← IXL SP ← SP - 2	* * X * X * * *	11 011 101 DD 11 100 101 E3	2	4	15	01 DE 10 HL 11 AF	
PUSH IY	(SP - 2) ← IYH (SP - 1) ← IYL SP ← SP - 2	* * X * X * * *	11 111 101 FD 11 100 101 E3	2	4	15		
POP qq	qqH ← (SP + 1) qqL ← (SP) SP ← SP + 2	* * X * X * * *	11 qq0 001	1	3	10		
POP IX	IXH ← (SP + 1) IXL ← (SP) SP ← SP + 2	* * X * X * * *	11 011 101 DD 11 100 001 E1	2	4	14		
POP IY	IYH ← (SP + 1) IYL ← (SP) SP ← SP + 2	* * X * X * * *	11 111 101 FD 11 100 001 E1	2	4	14		

dd es cualquiera de los pares de registros BC, DE, HL, SP.  
 qq es cualquiera (e los pares de registros AF, BC, DE, HL  
 (parh y (parl) se refieren al byte alto y al byte bajo dd  
 par; por ejemplo, BCL=C, AFH=A.

# Grupo de intercambio, transferencia de bloques y búsqueda de bloques

Código mnemotécnico	Operación simbólica	S	Z	Indicadores H P/V N C	Códigos 76 543 210 Hex	N.º de bytes	N.º de ciclos M	N.º de estados T	Comentarios
EX DE,HL EX AF,AF EXX	DE $\leftrightarrow$ HL AF $\leftrightarrow$ AF BC $\leftrightarrow$ BC DE $\leftrightarrow$ DE HL $\leftrightarrow$ HL	•	•	X • X • • • • •	11 101 011 EB	1	1	4	Intercambio del grupo de registros con el grupo alternativo
		•	•	X • X • • • • •	00 001 000 DB	1	1	4	
		•	•	X • X • • • • •	11 011 001 D9	1	1	4	
		•	•	X • X • • • • •	11 111 101 FD	1	1	4	
EX (SP),HL	H $\leftrightarrow$ (SP+1) L $\leftrightarrow$ (SP)	•	•	X • X • • • • •	11 100 011 E3	1	5	19	
EX (SP),IX	IX <sub>H</sub> $\leftrightarrow$ (SP+1) IX <sub>L</sub> $\leftrightarrow$ (SP)	•	•	X • X • • • • •	11 011 101 DD	2	6	23	
EX (SP),IY	IY <sub>H</sub> $\leftrightarrow$ (SP+1) IY <sub>L</sub> $\leftrightarrow$ (SP)	•	•	X • X • • • • •	11 100 011 E3	2	6	23	
LDF	(DE) $\leftarrow$ (HL) DE $\leftarrow$ DE+1 HL $\leftarrow$ HL+1 BC $\leftarrow$ BC-1	•	•	X 0 X ① 0 •	11 101 101 ED 10 100 000 A0	2	4	16	Carga (HL) en (DE), incrementa HL y DE y disminuye el contador BC
LDIR	(DE) $\leftarrow$ (HL) DE $\leftarrow$ DE+1 HL $\leftarrow$ HL+1 BC $\leftarrow$ BC-1 Repetir hasta que BC=0	•	•	X 0 X 0 0 •	11 101 101 ED 10 110 000 B0	2	5	21	Si BC $\neq$ 0 Si BC=0
LDD	(DE) $\leftarrow$ (HL) DE $\leftarrow$ DE-1 HL $\leftarrow$ HL-1 BC $\leftarrow$ BC-1	•	•	X 0 X ① 0 •	11 101 101 ED 10 101 000 A8	2	4	16	
LDDR	(DE) $\leftarrow$ (HL) DE $\leftarrow$ DE-1 HL $\leftarrow$ HL-1 BC $\leftarrow$ BC-1 Repetir hasta que BC=0	•	•	X 0 X 0 0 •	11 101 101 ED 10 111 000 B8	2	5	21	Si BC $\neq$ 0 Si BC=0
CPI	A $\leftarrow$ (HL) HL $\leftarrow$ HL+1 BC $\leftarrow$ BC-1	①	1	X 1 X ① 1 •	11 101 101 ED 10 100 001 A1	2	4	16	
CPIR	A $\leftarrow$ (HL) HL $\leftarrow$ HL+1 BC $\leftarrow$ BC-1 Repetir hasta que A=(HL) o BC=0	①	1	X 1 X ② 1 •	11 101 101 ED 10 110 001 B1	2	5	21	Si BC $\neq$ 0 y A $\neq$ (HL) Si BC=0 o A=(HL)
CPD	A $\leftarrow$ (HL) HL $\leftarrow$ HL-1 BC $\leftarrow$ BC-1	①	1	X 1 X ① 1 •	11 101 101 ED 10 101 001 A9	2	4	16	
CPDR	A $\leftarrow$ (HL) HL $\leftarrow$ HL-1 BC $\leftarrow$ BC-1 Repetir hasta que A=(HL) o BC=0	①	1	X 1 X ② 1 •	11 101 101 ED 10 111 001 B9	2	5	21	Si BC $\neq$ 0 y A $\neq$ (HL) Si BC=0 o A=(HL)

NOTAS ① a 0 si el resultado de BC-1 es 0; PV .

② P/V a 0 si se ha completado el recorrido; 1 a 1 en caso contrario.

③ Z a 1 si A=(HL); Z a 0 en caso contrario.

## Grupo aritmético y lógico de 8 bits

Código mnemotécnico	Operación simbólica	S	Z	H	P/V	N	C	Códigos 76 543 210 Hex	N.º de bytes	N.º de ciclos M	N.º de estados T	Comentarios
ADD A,r	A ← A+r	1	1	X	1	X	V	0 1 10 000 r	1	1	4	r Reg.
ADD A,n	A ← A+n	1	1	X	1	X	V	0 1 11 000 110	2	2	7	000 B
ADD A,(HL)	A ← A+(HL)	1	1	X	1	X	V	0 1 10 000 110	1	2	7	001 C
ADD A,(IX+d)	A ← A+(IX+d)	1	1	X	1	X	V	0 1 11 011 101 DD	3	5	19	010 D
								10 000 110				011 E
								→d→				100 H
ADD A,(IY+d)	A ← A+(IY+d)	1	1	X	1	X	V	0 1 11 111 101 FD	3	5	19	101 L
								10 000 110				111 A
								→d→				
ADC A,s	A ← A+s+CY	1	1	X	1	X	V	0 1 10 011				s puede ser un r, n,
SUB s	A ← A-s	1	1	X	1	X	V	1 1 10 011				(HL),m (IX+d), (IY+d)
SBC A,s	A ← A-s-CY	1	1	X	1	X	V	1 1 10 011				como para ADD; los
AND s	A ← A&s	1	1	X	1	X	P	0 0 10 011				códigos se forman como en
OR s	A ← A s	1	1	X	0	X	P	0 0 11 011				ADD, pero reemplazando
XOR s	A ← A⊕s	1	1	X	0	X	P	0 0 10 101				el 000 de ADD con los
CP s	A ← A-s	1	1	X	1	X	V	1 1 11 011				bits que se indican
INC r	r ← r+1	1	1	X	1	X	V	0 0 0 110 100	1	1	4	
INC (HL)	(HL) ← (HL)+1	1	1	X	1	X	V	0 0 11 011 101 DD	1	3	11	
INC (IX+d)	(IX+d) ← (IX+d)+1	1	1	X	1	X	V	0 0 11 011 101 DD	3	6	23	
								00 110 100				
								→d→				
INC (IY+d)	(IY+d) ← (IY+d)+1	1	1	X	1	X	V	0 1 11 111 101 FD	3	6	23	
								00 110 100				
								→d→				
DEC m	m ← m-1	1	1	X	1	X	V	1 1 10 101				m es un r, (HL), (IX+d),
								101				(IY+d) como para INC;
												DEC tiene igual formato y
												estados que INC; el código
												se forma como en INC, pero
												reemplazando el 100 de INC por
												101

## Grupo aritmético y de control de aplicación general

Código mnemotécnico	Operación simbólica	S	Z	H	P/V	N	C	Códigos 76 543 210 Hex	N.º de bytes	N.º de ciclos M	N.º de estados T	Comentarios
DAA	Ajusta el acumulador para operaciones de suma y resta en BCD <sup>(1)</sup> con operandos BCD	1	1	X	1	X	P	0 0 100 111 2F	1	1	4	Ajuste decimal del acumulador
CPL	A ← $\bar{A}$	*	*	X	1	X	*	0 0 101 111 2F	1	1	4	Complementa (a 1) el acumulador
NEG	A ← 0-A	1	1	X	1	X	V	1 1 101 101 ED 01 000 100 44	2	2	8	Cambia de signo del acumulador (complemento a 2)
CCF	CY ← $\bar{CY}$	*	*	X	X	X	*	0 1 00 111 111 3F	1	1	4	Complementa el indicador de arrastre
SCF	CY ← 1	*	*	X	0	X	*	0 1 00 110 111 37	1	1	4	Pone a 1 el indicador de arrastre
NOP	No operación	*	*	X	X	X	*	0 0 000 000 00	1	1	4	
HALT	Detiene el Z80	*	*	X	X	X	*	0 1 110 110 76	1	1	4	
DI*	IFF ← 0	*	*	X	X	X	*	1 1 110 011 F3	1	1	4	
EI*	IFF ← 1	*	*	X	X	X	*	1 1 111 011 F3	1	1	4	
IM 0	Selecciona modo 0 para las interrupciones	*	*	X	X	X	*	1 1 101 101 ED 01 000 110 46	2	2	8	
IM 1	Selecciona modo 1 para las interrupciones	*	*	X	X	X	*	1 1 101 101 ED 01 010 110 56	2	2	8	
IM 2	Selecciona modo 2 para las interrupciones	*	*	X	X	X	*	1 1 101 101 ED 01 011 110 5E	2	2	8	

NOTAS: (1) BCD: decimal codificado en binario.

IFF representa la báscula de habilitación de interrupciones.

CY representa el indicador de arrastre.














\* indica que las interrupciones no son examinadas al final de DI o EI.

## Grupo aritmético de 16 bits

Código mnemotécnico	Operación simbólica	Indicadores S Z H P/V N C	Códigos 76 543 210 Hex	N.º de bytes	N.º de ciclos M	N.º de estados T	Comentarios
ADD HL,ss	HL ← HL + ss	• • X X X • • 0 1	00 ss1 001	1	3	11	ss Par
ADC HL,ss	HL ← HL + ss + CY	1 1 X X X V 0 1	11 101 101 ED 01 ss1 010	2	4	15	00 BC 01 DE 10 HL 11 SP
SBC HL,ss	HL ← HL - ss - CY	1 1 X X X V 1 1	11 101 101 ED 01 ss0 010	2	4	15	pp Par
ADD IX,pp	IX ← IX + pp	• • X X X • • 0 1	11 011 101 DD 01 ppl 001	2	4	15	00 BC 01 DE 10 IX 11 SP
ADD IY,rr	IY ← IY + rr	• • X X X • • 0 1	11 111 101 FD 00 rrr 001	2	4	15	rr Par
							00 BC 01 DE 10 IY 11 SP
INC ss	ss ← ss + 1	• • X • X • • • • 00	ss0 011	1	1	6	
INC IX	IX ← IX + 1	• • X • X • • • • 11	011 101 DD 00 100 011 23	2	2	10	
INC IY	IY ← IY + 1	• • X • X • • • • 11	111 101 FD 00 100 011 23	2	2	10	
DEC ss	ss ← ss - 1	• • X • X • • • • 00	ss1 011	1	1	6	
DEC IX	IX ← IX - 1	• • X • X • • • • 11	011 101 DD 00 101 011 23	2	2	10	
DEC IY	IY ← IY - 1	• • X • X • • • • 11	111 101 FD 00 101 011 23	2	2	10	

NOTAS: SS es cualquiera de los pares de registros BC, DE, HL, SP.  
pp es cualquier de los pares de registros BC, DE, IX, SP.  
rr es cualquiera de los pares de registros BC, DE, IY, SP.

## Grupo de rotación y desplazamiento

Código mnemotécnico	Operación simbólica	Indicadores S Z H P/V N C	Códigos 76 543 210 Hex	N.º de bytes	N.º de ciclos M	N.º de estados T	Comentarios
RLCA		• • X 0 X • 0 1	00 000 111 07	1	1	4	Rotación circular a la izquierda del acumulador
RLA		• • X 0 X • 0 1	00 010 111 17	1	1	4	Rotación a la izquierda del acumulador
RRCa		• • X 0 X • 0 1	00 001 111 0F	1	1	4	Rotación circular a la derecha del acumulador
RRA		• • X 0 X • 0 1	00 011 111 1F	1	1	4	Rotación a la derecha del acumulador
RLC r		1 1 X 0 X P 0 1	11 001 011 CB 00 000 r	2	2	8	Rotación circular a la izquierda del registro r
RLC (HL)		1 1 X 0 X P 0 1	11 001 011 CB 00 000 110	2	4	15	r Reg.
RLC ((X+d))		1 1 X 0 X P 0 1	11 011 101 DB 11 001 011 CB -d-	4	6	23	000 B 001 C 010 D 011 E 100 H 101 L 111 A
RLC ((Y+d))		1 1 X 0 X P 0 1	11 111 101 FB 11 001 011 CH -d-	4	6	23	Todas con el mismo formato y estados que las RLC; el código se forma como en las RLC, pero reemplazando el 0001 de RLC por el propio código de la operación.
RL m		1 1 X 0 X P 0 1	00 000 110 010				
RRC m		1 1 X 0 X P 0 1	001				
RR m		1 1 X 0 X P 0 1	011				
SLA m		1 1 X 0 X P 0 1	100				
SRA m		1 1 X 0 X P 0 1	101				
SRL m		1 1 X 0 X P 0 1	111				
RLD		1 1 X 0 X P 0 •	11 101 101 ED 01 101 111 6F	2	5	18	Rotación de las cifras DCB, a la derecha y a izquierda, entre la mitad baja del acumulador y la posición (HL); no afectan a la mitad alta del acumulador.
RRD		1 1 X 0 X P 0 •	11 101 101 ED 01 100 111 6F	2	5	18	

## Grupo de manipulación de bits

Código mnemotécnico	Operación simbólica	Indicadores					Códigos 76 543 210 Hex	N.º de bytes	N.º de ciclos M	N.º de estados T	Comentarios	
		S	Z	H	P/V	N C						
BIT $b, r$	$Z \leftarrow r_b$	X	1	X	1	X	0 *	11 001 011 CB	2	2	8	r Reg.
								01 b r				000 B
BIT $b, (HL)$	$Z \leftarrow (HL)_b$	X	1	X	1	X	0 *	11 001 011 CB	2	3	12	001 C
								01 b 110				010 D
BIT $b, (IX+d)_b$	$Z \leftarrow ((IX+d)_b)$	X	1	X	1	X	0 *	11 011 101 DD	4	5	20	011 E
								11 001 011 CB				100 H
								$\leftarrow d \rightarrow$				101 L
								01 b 110				111 A
BIT $b, (IY+d)_b$	$Z \leftarrow ((IY+d)_b)$	X	1	X	1	X	0 *	11 111 101 FD	4	5	20	Bit comprob.
								11 001 011 CB				b
								$\leftarrow d \rightarrow$				000 0
								01 b 110				001 1
												010 2
												011 3
												100 4
												101 5
												110 6
												111 7
SET $b, r$	$r_b \leftarrow 1$	*	*	X	*	X	*	11 001 011 CB	2	2	8	
								11 b r				
SET $b, (HL)$	$(HL)_b \leftarrow 1$	*	*	X	*	X	*	11 001 011 CB	2	4	15	
								11 b 110				
SET $b, (IX+d)$	$(IX+d)_b \leftarrow 1$	*	*	X	*	X	*	11 011 101 DD	4	6	23	
								11 001 011 CB				
								$\leftarrow d \rightarrow$				
								11 b 110				
SET $b, (IY+d)$	$(IY+d)_b \leftarrow 1$	*	*	X	*	X	*	11 111 101 FD	4	6	23	
								11 001 012 CB				
								$\leftarrow d \rightarrow$				
								11 b 110				
RES $b, m$	$m_b \leftarrow 0$ $m \oplus r, (HL), (IX+d), (IY+d)$	*	*	X	*	X	*	11 110				El código se forma como en las SET b,m, pero reemplazando 11 por 10; indicadores y estados como para SET
								10				

NOTA: mh representa el bit b (0 a 7) del registro o la posi-

## Grupo de salto

Código mnemotécnico	Operación simbólica	Indicadores S Z H P/V N C	Códigos 76 543 210 Hex	N.º de bytes	N.º de ciclos M	N.º de estados T	Comentarios
JP nn	PC ← nn	* * X * X * * *	11 000 011 C3 ←n→ ←n→	3	3	10	
JP cc,nn	Si la condición cc es cierta, PC ← nn; si no, se continúa	* * X * X * * *	11 cc 010 ←n→ ←n→	3	3	10	cc Condición 000 NZ no cero 001 Z cero 010 NC no acarriate 011 C acarriate 100 PO paridad impar 101 PE paridad par 110 P signo positivo 111 M signo negativo
JR e	PC ← PC + e	* * X * X * * *	00 011 000 18 ←e-2→	2	3	12	
JR C,e	Si C=0, se continúa	* * X * X * * *	00 111 000 38 ←e-2→	2	2	7	Si no se cumple la condición
JR NC,e	Si C=1, PC ← PC + e Si C=0, se continúa	* * X * X * * *	00 110 000 30 ←e-2→	2	2	7	Si se cumple la condición Si no se cumple la condición
JP Z,c	Si C=0, PC ← PC + c Si Z=0, se continúa	* * X * X * * *	00 101 000 28 ←c-2→	2	3	12	Si se cumple la condición Si no se cumple la condición
JR NZ,e	Si Z=1, PC ← PC + e Si Z=0, se continúa	* * X * X * * *	00 100 000 20 ←e-2→	2	3	12	Si se cumple la condición Si no se cumple la condición
JP (HL)	PC ← HL	* * X * X * * *	11 101 001 E9	1	1	4	Si se cumple la condición
JP (IX)	PC ← IX	* * X * X * * *	11 011 101 DD 11 101 001 E9	2	2	6	
JP (IY)	PC ← IY	* * X * X * * *	11 111 101 FD 11 101 001 E9	2	2	6	
DJNZ, e	B ← B - 1 Si B=0, se continúa Si B≠0, PC ← PC + e	* * X * X * * *	00 010 000 10 ←e-2→	2	2	8	Si B=0
				2	3	13	Si B≠0

NOTAS: C es la magnitud del salto relativo; es un número con signo del intervalo -126 ... 129 en el código figura c-2 para provocar un salto de de sumarle el desplazamiento

## Grupo de llamada y retorno

Código mnemotécnico	Operación simbólica	Indicadores S Z H P/V N C	Códigos 76 843 210 Hex	N.º de bytes	N.º de ciclos M	N.º de estados T	Comentarios
CALL nn	(SP-1) ← PC <sub>11</sub> (SP-2) ← PC <sub>11</sub> PC ← nn	* * X * X * * *	11 001 101 CD ← n → ← n →	3	3	17	
CALL cc,nn	Si la condición cc es falsa, se continúa; si es cierta, como CALL nn	* * X * X * * *	11 cc 100 ← n → ← n →	3	3	10	Si cc es falsa
RET	PC <sub>11</sub> ← (SP) PC <sub>11</sub> ← (SP+1)	* * X * X * * *	11 001 001 C9	1	3	10	
RET cc	Si la condición cc es falsa, se continúa; si no, como RET	* * X * X * * *	11 cc 000 1 3	1	3	11	Si cc es falsa Si cc es cierta cc Condición
RETI	Retorno de una interrupción	* * X * X * * *	11 101 101 ED 01 001 101 4D	2	4	14	000 NZ no cero 001 Z cero 010 NC no arrastre 011 C arrastre 100 PD paridad impar 101 PE paridad par 110 P signo positivo 111 M signo negativo
RETN <sup>(1)</sup>	Retorno de una inte- rrupción no enmascarable	* * X * X * * *	11 101 101 ED 01 000 101 45	2	4	14	
RST n	(SP-1) ← PC <sub>11</sub> (SP-2) ← PC <sub>11</sub> PC <sub>11</sub> ← 0 PC <sub>11</sub> ← 0	* * X * X * * *	11 1 111	1	3	11	1 p 000 03H 001 05H 010 10H 011 16H 100 20H 101 28H 110 30H 111 36H

NOTAS: <sup>(1)</sup> RETN carga 1FF<sub>2</sub>-1FF<sub>1</sub>.



## Grupo de entrada y salida

Código mnemotécnico	Operación simbólica	Indicadores S Z H P/V N C	Códigos 76 543 210 Hex	N.º de bytes	N.º de ciclos M	N.º de estados T	Comentarios
IN A,(n)	A ← (n)	• • X • X • • •	11 011 011 DB ← n →	2	3	11	n a A <sub>0</sub> -A <sub>7</sub> Acumulador a A <sub>2</sub> -A <sub>15</sub> C a A <sub>0</sub> -A <sub>7</sub> B a A <sub>8</sub> -A <sub>15</sub>
IN r,(C)	r ← (C) Si r ≠ 110, sólo quedan afectados los indicadores	1 1 X 1 X P 0 •	11 101 101 ED 01 r 000	2	3	12	
INI	(HL) ← (C) B ← B - 1	X 1 X X X X 1 X	11 101 101 ED 10 100 010 A2	2	4	16	C a A <sub>0</sub> -A <sub>7</sub> B a A <sub>8</sub> -A <sub>15</sub>
INIR	HL ← HL + 1 (HL) ← (C) B ← B - 1 HL ← HL + 1 Se repite hasta que B = 0	X 1 X X X X 1 X	11 101 101 ED 10 110 010 B2	2	5 (Si B ≠ 0) 4 (Si B = 0)	21 16	C a A <sub>0</sub> -A <sub>7</sub> B a A <sub>8</sub> -A <sub>15</sub>
IND	(HL) ← (C) B ← B - 1 HL ← HL - 1	X 1 X X X X 1 X	11 101 101 ED 10 101 010 A4	2	4	16	C a A <sub>0</sub> -A <sub>7</sub> B a A <sub>8</sub> -A <sub>15</sub>
INDR	(HL) ← (C) B ← B - 1 HL ← HL - 1 Se repite hasta que B = 0	X 1 X X X X 1 X	11 101 101 ED 10 111 010 B4	2	5 (Si B ≠ 0) 4 (Si B = 0)	21 16	C a A <sub>0</sub> -A <sub>7</sub> B a A <sub>8</sub> -A <sub>15</sub>
OUT (n),A	(n) ← A	• • X • X • • •	11 010 011 D3 ← n →	2	3	11	n a A <sub>0</sub> -A <sub>7</sub> Acumulador a A <sub>8</sub> -A <sub>15</sub> C a A <sub>0</sub> -A <sub>7</sub> B a A <sub>8</sub> -A <sub>15</sub>
OUT (C),r	(C) ← r	• • X • X • • •	11 101 101 ED 01 r 001	2	3	12	
OUTI	(C) ← (HL) B ← B - 1 HL ← HL + 1	X 1 X X X X 1 X	11 101 101 ED 10 100 011 A3	2	4	16	C a A <sub>0</sub> -A <sub>7</sub> B a A <sub>8</sub> -A <sub>15</sub>
OTIR	(C) ← (HL) B ← B - 1 HL ← HL + 1 Se repite hasta que B = 0	X 1 X X X X 1 X	11 101 101 ED 10 110 011 B3	2	5 (Si B ≠ 0) 4 (Si B = 0)	21 16	C a A <sub>0</sub> -A <sub>7</sub> B a A <sub>8</sub> -A <sub>15</sub>
OUTD	(C) ← (HL) B ← B - 1 HL ← HL - 1	X 1 X X X X 1 X	11 101 101 ED 10 101 011 AB	2	4	16	C a A <sub>0</sub> -A <sub>7</sub> B a A <sub>8</sub> -A <sub>15</sub>
OTDR	(C) ← (HL) B ← B - 1 HL ← HL - 1 Se repite hasta que B = 0	X 1 X X X X 1 X	11 101 101 ED 10 111 011	2	5 (Si B ≠ 0) 4 (Si B = 0)	21 16	C a A <sub>0</sub> -A <sub>7</sub> B a A <sub>8</sub> -A <sub>15</sub>

NOTAS: ① Z a 1 si B = 1 = 0; si no, Z a 0.

## Resumen de los indicadores afectados

Instrucciones	D <sub>7</sub> Indicadores D <sub>0</sub>						Comentarios				
	S	Z	H	P/V	N	C					
ADD A,s; ADC A,s	1	1	1	1	X	V	0	1	Suma o suma con arrastre de 8 bits.		
SUB s; SBC A,s; CP s; NEG	1	1	1	1	1	X	V	1	1	Resta o resta con arrastre de 8 bits. Comparación. Cambio de signo del acumulador.	
AND s	1	1	1	1	X	P	0	0	0	Operaciones lógicas.	
OR s; XOR s	1	1	1	1	X	P	0	0	0		
INC s	1	1	1	1	1	X	V	0	•	Incremento para 8 bits.	
DEC s	1	1	1	1	1	X	V	1	•	Disminución para 8 bits.	
ADD DD,s	•	•	•	•	X	•	•	0	1	Suma de 16 bits.	
ADC HL,s	1	1	1	1	X	X	V	1	1	Suma con arrastre de 16 bits.	
SBC HL,s	1	1	1	1	X	X	V	1	1	Resta con arrastre de 16 bits.	
RLA; RLCA; RRA; RRCA	•	•	•	•	X	0	X	•	•	Rotación del acumulador.	
RL m; RLC m; RR m;	1	1	1	1	X	0	X	P	0	1	Rotaciones y desplazamientos
RRC m; SLA m; SRA m;											
SRL m											
RLD; RRD	1	1	1	1	X	0	X	P	0	•	Rotación de las cifras decimales a la izquierda y a la derecha
DAA	1	1	1	1	1	X	P	•	1	•	Ajuste decimal del acumulador
CPL	•	•	•	•	1	X	1	•	•	•	Complemento del acumulador
SCF	•	•	•	•	X	0	X	•	0	1	Puesta a 1 del indicador de arrastre
CCF	•	•	•	•	X	X	X	•	0	1	Complemento del indicador de arrastre
IN r,(C)	1	1	1	1	X	0	X	P	0	•	Entrada de un registro
INI; IND; OUT; OUTD	X	1	1	1	X	X	X	1	•	•	Entrada y salida de bloques. Z a 0 si B≠0; Z a 1 en caso contrario.
INIR; INDR; OTIR; OTDR	X	1	1	1	X	X	X	X	1	•	
LDI; LDD	X	1	1	1	X	0	X	1	0	•	Transferencia de bloques. P/V a 1 si BC≠0; P/V a 0 en caso contrario.
LDIR; LDDR	X	1	1	1	X	0	X	0	0	•	
CP; CPir; CPD; CPDR	X	1	1	1	X	X	X	1	1	•	Búsqueda en bloques. Z a 1 si A=(HL); Z a 0 en caso contrario. P/V a 1 si BC≠0; P/V a 0 en caso contrario.
LD A,I; LD A,R	1	1	1	1	X	0	X	IFF	0	•	El contenido de la báscula de habilitación de interrupciones se carga en el indicador P/V.
BIT b,s	X	1	1	1	1	X	X	0	•	•	El bit b del registro o posición s se carga en el indicador Z.

## Notaciones empleadas

Símbolo	Operación	Símbolo	Operación
S	Indicador de signo. S a 1 cuando el bit más significativo del resultado es 1.	1	El indicador queda afectado según sea el resultado de la operación.
Z	Indicador de cero. Z a 1 cuando el resultado de la operación es 0.	•	La instrucción no afecta al indicador.
P/V	Indicador de paridad (P) o de sobrepasamiento (V); los dos comparten el mismo bit. En las operaciones lógicas tiene el significado de paridad; en las operaciones aritméticas con signo, tiene el significado de sobrepasamiento. Como indicador de paridad, P/V se pone a 1 con paridad par, y a 0 con paridad impar. Como indicador de sobrepasamiento, P/V se pone a 1 cuando se produce sobrepasamiento.	0	La operación coloca a 0 el indicador.
H	Indicador de semiarresto. H a 1 cuando en una suma hay acarreo desde el bit 3 y cuando en una resta no hay acarreo negativo desde el bit 4 del acumulador.	1	La operación coloca a 1 el indicador.
N	Indicador de suma/resta. N a 1 cuando la operación anterior es una resta.	X	Puede afectar el indicador en forma impredecible.
H & N	H y N se usan junto con la instrucción DAA (ajuste decimal del acumulador) para efectuar la corrección del resultado de una operación de suma o resta en formato BCD.	V	El indicador P/V queda afectado en el sentido de sobrepasamiento, según sea el resultado de la operación.
C	Indicador de arrastre. C a 1 cuando se produce un arrastre en el bit más significativo al efectuar una operación.	P	El indicador P/V queda afectado de acuerdo con la paridad del resultado.
		r	Cualquiera de los registros A, B, C, D, E, H, L.
		s	Registro, número de 8 bits o posición direccionada de cualquiera de las maneras que admita la instrucción.
		m	Registro o posición direccionada de cualquiera de las maneras que admita la instrucción.
		ss, dd,	Un par de registros que sea admisible para la operación
		qq, pp,	
		rr	
		ix	Cualquiera de los dos registros índices: IX, IY.
		k	Registro de regeneración de memoria.
		a	Un valor de 8 bits en el intervalo (0...255).
		nn	Un valor de 16 bits en el intervalo (0...65535).

## Apéndice B

```
1000 REM APENDICE B
1010 REM CARGADOR HEX
1020 MODE 1 : PAPER 0 : PEN 1
1030 ER% = 1 : L% = 4
1040 PAPER 0 : PEN 2: PRINT "PONER HIME
M EN";
1050 A% = 0 : B = 0 : GOSUB 1280
1060 IF B > 43900 OR B < 2000 THEN ER% =
1 : GOTO 1250
1070 MM = 43903 : MEMORY B
1080 PAPER 2 : PEN 0 : PRINT " HIMEM
EN "; HEX$(HIMEM,4); " HEX"
1090 L% = 4
1100 PAPER 0 : PEN 2 : PRINT "DIRECCION
INICIAL";
1110 A% = 0 : B = 0 : GOSUB 1280
1120 IF B <= HIMEM THEN ER% = 2 : GOTO 1
250
1130 IF B > 43903 THEN ER% = 5 : GOTO 12
50
1140 INIC = B : PAPER 2 : PEN 0 : PRINT
"DIR INIC "; HEX$(B,4); " HEX" : P
APER 3 : PEN 1 : PRINT "TECLEE LOS DATOS
" : PAPER 0
1150 IDIR = B
1160 ADIR = IDIR : SUMA = 0
1170 L% = 2
1180 WHILE ADIR < IDIR + 10
1190 GOSUB 1270 : POKE ADIR, B : PEN 2 :
PRINT HEX$(ADIR,4), HEX$(B,2) : PEN 1 :
SUMA = SUMA + B : ADIR = ADIR + 1 : IF A
DIR >= MM - 2 THEN ADIR = IDIR + 20
1200 WEND : IF ADIR = IDIR + 20 THEN ER%
= 4 : GOTO 1250
1210 PAPER 3 : PRINT "TECLEE LA SUMA ";
: PAPER 0 : L% = 4 : GOSUB 1270
1220 IF SUMA <> B THEN ER% = 3 : GOTO 1250

1230 IF FIN = 1 THEN PEN 2 : PAPER 3 : P
RINT " TERMINADO" : PEN 1 : INPUT " MAS?
S/N "; A$ : PAPER 0 : A$ = UPPER$(A$) :
IF ASC(A$) = 83 THEN FIN = 0 : GOTO 108
0 ELSE END
```

```

1240 IDIR = ADIR : PEN 0 : PAPER 2 : PRI
NT "SUMA "; HEX$(B,4); " CORRECTA ; TECL
EE MAS DATOS" : PEN 1 : PAPER 0 : GOTO 1
160
1250 RESTORE 1390 : PEN 3 : PAPER 1 : FO
R N% = 1 TO ER% : READ D$ : NEXT : PRINT
D$; ", TECLEE OTRA VEZ"; CHR$(7)
1260 PEN 1 : PAPER 0 : ON ER% GOTO 1030,
1090,1160,1030,1090
1270 A% = 0 : B = 0 : PEN 1
1280 INPUT IN$ : PRINT CHR$(11); : IN$ =
UPPER$(IN$) : IF IN$ = "END" THEN 1370
1290 IF LEN(IN$) <> L% THEN 1360
1300 FOR N% = 1 TO L%
1310 A$ = MID$(IN$,N%,1) : IF A$ > "F" 0
R A$ < "0" OR ( A$ > "9" AND A$ < "A" )
THEN 1360
1320 IF A$ > "9" THEN A% = ASC(A$) : A%
= ( A% AND &F) + 9 ELSE A% = VAL(A$)
1330 IF N% <> L% THEN B = B + ( A% * 16
^ ( L% - N% )) ELSE B = B + A%
1340 NEXT
1350 RETURN
1360 PEN 3: PAPER 1 : PRINT "NO ES VALI
DO ,TECLEE OTRA VEZ"; CHR$(7) : PEN 1 :
PAPER 0 : GOTO 1270
1370 REM END
1380 FIN = 1 : GOTO 1210
1390 DATA DEMASIADO ALTO O BAJO , AREA D
E LA MEMORIA NO PROTEGIDA, LA SUMA NO C
OINCIDE; DEBE REEMPRENDER LA INTRODUCCIO
N A PARTIR DE LA ULTIMA SUMA , MEMORIA C
OMPLETA , DEMASIADO ALTO

```

## Apéndice C

### Conversión de HEX a DECIMAL para el byte más significativo

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	0	256	512	768	1024	1280	1536	1792	2048	2304	2560	2816	3072	3328	3584	3840
1	4096	4352	4608	4864	5120	5376	5632	5888	6144	6400	6656	6912	7168	7424	7680	7936
2	8192	8448	8704	8960	9216	9472	9728	9984	10240	10496	10752	11008	11264	11520	11776	12032
3	12288	12544	12800	13056	13312	13568	13824	14080	14336	14592	14848	15104	15360	15616	15872	16128
4	16384	16640	16896	17152	17408	17664	17920	18176	18432	18688	18944	19200	19456	19712	19968	20224
5	20480	20736	20992	21248	21504	21760	22016	22272	22528	22784	23040	23296	23552	23808	24064	24320
6	24576	24832	25088	25344	25600	25856	26112	26368	26624	26880	27136	27392	27648	27904	28160	28416
7	28672	28928	29184	29440	29696	29952	30208	30464	30720	30976	31232	31488	31744	32000	32256	32512
8	32768	33024	33280	33536	33792	34048	34304	34560	34816	35072	35328	35584	35840	36096	36352	36608
9	36864	37120	37376	37632	37888	38144	38400	38656	38912	39168	39424	39680	39936	40192	40448	40704
A	40960	41216	41472	41728	41984	42240	42496	42752	43008	43264	43520	43776	44032	44288	44544	44800
B	45056	45312	45568	45824	46080	46336	46592	46848	47104	47360	47616	47872	48128	48384	48640	48896
C	49152	49408	49664	49920	50176	50432	50688	50944	51200	51456	51712	51968	52224	52480	52736	52992
D	53248	53504	53760	54016	54272	54528	54784	55040	55296	55552	55808	56064	56320	56576	56832	57088
E	57344	57600	57856	58112	58368	58624	58880	59136	59392	59648	59904	60160	60416	60672	60928	61184
F	61440	61696	61952	62208	62464	62720	62976	63232	63488	63744	64000	64256	64512	64768	65024	65280



## Apéndice D

### Conversión de HEX a DECIMAL para el byte menos significativo

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
2	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
3	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
4	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79
5	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95
6	96	97	98	99	100	101	102	103	104	105	106	107	108	109	110	111
7	112	113	114	115	116	117	118	119	120	121	122	123	124	125	126	127
8	128	129	130	131	132	133	134	135	136	137	138	139	140	141	142	143
9	144	145	146	147	148	149	150	151	152	153	154	155	156	157	158	159
A	160	161	162	163	164	165	166	167	168	169	170	171	172	173	174	175
B	176	177	178	179	180	181	182	183	184	185	186	187	188	189	190	191
C	192	193	194	195	196	197	198	199	200	201	202	203	204	205	206	207
D	208	209	210	211	212	213	214	215	216	217	218	219	220	221	222	223
E	224	225	226	227	228	229	230	231	232	233	234	235	236	237	238	239
F	240	241	242	243	244	245	246	247	248	249	250	251	252	253	254	255

HEX	DEC	SIN	HEX	DEC	BIN
0	0	0000	8	8	1000
1	1	0001	9	9	1001
2	2	0010	A	10	1010
3	3	0011	B	11	1011
4	4	0100	C	12	neo
5	5	0101	D	13	1101
6	6	0110	E	14	1110
7	7	0111	F	15	1111





# Apéndice E

## Conversión de HEX en complemento a 2 a DECIMAL

### Para el byte más significativo

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	0	256	512	768	1024	1280	1536	1792	2048	2304	2560	2816	3072	3328	3584	3840
1	4096	4352	4608	4864	5120	5376	5632	5888	6144	6400	6656	6912	7168	7424	7680	7936
2	8192	8448	8704	8960	9216	9472	9728	9984	10240	10496	10752	11008	11264	11520	11776	12032
3	12288	12544	12800	13056	13312	13568	13824	14080	14336	14592	14848	15104	15360	15616	15872	16128
4	16384	16640	16896	17152	17408	17664	17920	18176	18432	18688	18944	19200	19456	19712	19968	20224
5	20480	20736	20992	21248	21504	21760	22016	22272	22528	22784	23040	23296	23552	23808	24064	24320
6	24576	24832	25088	25344	25600	25856	26112	26368	26624	26880	27136	27392	27648	27904	28160	28416
7	28672	28928	29184	29440	29696	29952	30208	30464	30720	30976	31232	31488	31744	32000	32256	32512
8	-22720	-32512	-32256	-32000	-31744	-31488	-31232	-30976	-30720	-30464	-30208	-29952	-29696	-29440	-29184	-28928
9	-28672	-28416	-28160	-27904	-27648	-27392	-27136	-26880	-26624	-26368	-26112	-25856	-25600	-25344	-25088	-24832
A	-24576	-24320	-24064	-23808	-23552	-23296	-23040	-22784	-22528	-22272	-22016	-21760	-21504	-21248	-20992	-20736
B	-20480	-20224	-19968	-19712	-19456	-19200	-18944	-18688	-18432	-18176	-17920	-17664	-17408	-17152	-16896	-16640
C	-16384	-16128	-15872	-15616	-15360	-15104	-14848	-14592	-14336	-14080	-13824	-13568	-13312	-13056	-12800	-12544
D	-12288	-12032	-11776	-11520	-11264	-11008	-10752	-10496	-10240	-9984	-9728	-9472	-9216	-8960	-8704	-8448
E	-8192	-7936	-7680	-7424	-7168	-6912	-6656	-6400	-6144	-5888	-5632	-5376	-5120	-4864	-4608	-4352
F	-4096	-3840	-3584	-3328	-3072	-2816	-2560	-2304	-2048	-1792	-1536	-1280	-1024	-768	-512	-256

Para calcular el valor decimal de un número negativo de 16 bits, se debe sumar al valor del byte más significativo que proporciona esta tabla (y que será negativo) el valor del byte menos significativo interpretado sin signo (luego positivo).

## Para el byte menos significativo

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
2	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
3	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
4	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79
5	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95
6	96	97	98	99	100	101	102	103	104	105	106	107	108	109	110	111
7	112	113	114	115	116	117	118	119	120	121	122	123	124	125	126	127
8	-128	-127	-126	-125	-124	-123	-122	-121	-120	-119	-118	-117	-116	-115	-114	-113
9	-112	-111	-110	-109	-108	-107	-106	-105	-104	-103	-102	-101	-100	-99	-98	-97
A	-96	-95	-94	-93	-92	-91	-90	-89	-88	-87	-86	-85	-84	-83	-82	-81
B	-80	-79	-78	-77	-76	-75	-74	-73	-72	-71	-70	-69	-68	-67	-66	-65
C	-64	-63	-62	-61	-60	-59	-58	-57	-56	-55	-54	-53	-52	-51	-50	-49
D	-48	-47	-46	-45	-44	-43	-42	-41	-40	-39	-38	-37	-36	-35	-34	-33
E	-32	-31	-30	-29	-28	-27	-26	-25	-24	-23	-22	-21	-20	-19	-18	-17
F	-16	-15	-14	-13	-12	-11	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1

## Apéndice F

### Mapa de pantalla del Amstrad

El mapa de pantalla del Amstrad CPC464 presenta cierta complejidad. Por una parte, puede cambiar la dirección en que comienza. Pero, además, resulta que un punto (*pixel*.) puede estar representado por bits diferentes, según el modo de pantalla que se seleccione.

La pantalla ocupa siempre 16K de memoria. Lo normal es que comience en la posición C000h (49152), aunque también se puede hacer por programa que comience en 4000h (16384). Para lo que sigue vamos a suponer que comienza en C000h, ya que es poco probable que usted necesite cambiar esta dirección.

La pantalla está siempre formada por 200 líneas de un punto de altura. Cada una de estas líneas ocupa 80 bytes consecutivos de la memoria, que comenzarán en alguna dirección que será C000h más un múltiplo de 80. Cada carácter ocupa 8 por 8 puntos. En el modo 2 de pantalla cada punto se corresponde con un bit: si el bit está a 1 el punto será iluminado con el color de la tinta 1 y, si está a 0, con la tinta 0.

Mientras la pantalla no se haya movido hacia arriba, su esquina superior izquierda corresponderá a C000h. Los primeros 80 bytes forman la línea de arriba, pero los segundos 80 bytes no forman la segunda, sino la línea de arriba de la segunda fila de caracteres, que es la novena línea de puntos. Los 80 bytes siguientes corresponden a la línea 17, y así hasta completar las 25 filas de caracteres. Sólo entonces comienza la segunda línea de puntos. En la figura de la página siguiente se muestran las direcciones del primero y último byte de cada línea de puntos para las primeras 24 líneas (en la posición inicial de la pantalla).

El sistema operativo proporciona rutinas que permiten calcular la dirección de un carácter o de un punto. Las direcciones de estas rutinas se dan en el Apéndice G.

En los modos 1 y 0 se mantiene el orden de los bytes para las líneas de puntos de la pantalla, pero cambia la forma en que un byte representa determinados puntos. En modo 1 cada byte almacena la información de cuatro puntos y en modo cero de sólo dos puntos. El orden de la representación no es directo en el interior de cada byte. Un byte representa los puntos en las formas siguientes:

Linea n.º	Dirección		Línea n.º	Dirección	
	Izda.	Dcha.		Izda.	Dcha.
1	C000	C04F	13	E050	E09F
2	C800	C84F	14	E850	E89F
3	D000	D04F	15	F050	F09F
4	DB00	DS4F	16	FB50	F89F
5	E000	E04F	17	C0A0	C0DF
6	E800	EB4F	18	C8A0	C8DF
7	F000	F04F	19	D0A0	D0DF
8	F800	F84F	20	D8A0	D8DF
9	C050	C09F	21	E0A0	E0DF
10	C850	ca?F	22	E8A0	E80F
11	D050	D09F	23	F0A0	F0DF
12	D850	D89F	24	FBA0	F8DF

Modo 1; puntos de izquierda a derecha  
bits 3 y 7 2 y 6 1 y 5 0 y 4

Modo 0; puntos de izquierda a derecha  
bits 1,5,3 y 7 0,4,2 y 6

Los bits de cada punto están dados en orden de significación decreciente respecto de la forma en que componen el código binario que representa el número de tinta de cada punto.

Por ejemplo, la dirección C000h cargada con 01010011b representa en modo 1 cuatro puntos, de los colores 0, 1, 2 y 3 respectivamente.

En modo 0, el mismo byte representaría dos puntos de tintas 8 y 13. Para obtener en este modo cuatro puntos de tintas 0, 1, 2 y 3, se requerirían dos bytes cargados con

01000000b 01001100b

Cuando la pantalla se desplaza, cambia la dirección del byte de la esquina superior izquierda. Esta dirección puede oscilar de C000+80 a 80\*25 MOD 2048. Afortunadamente, existen rutinas del firmware que establecen la dirección en que comienza la pantalla (véase el apéndice G).

## Apéndice G

### Dirección de las rutinas más usuales sistema operativo

En los programas del libro hemos utilizado algunas de las rutinas del sistema operativo. La que hemos utilizado con la etiqueta GETKEY (BB18h) es la que suele llamarse 'wait key'; corresponde al área del firmware llamada 'KEY MANAGER', que agrupa una serie de rutinas para el control del teclado. La que se utilizaba con la etiqueta PRINT (BB5Ah) es la rutina 'text output' que corresponde al área 'TEXT VDU' que agrupa rutinas relativas a la pantalla de texto. Hay otras siete áreas, que llevan los nombres de 'GRAPHICS VDU', 'SCREEN PACK', 'CASSETTE MANAGER', 'SOUND MANAGER', 'KERNEL', 'MACHINE PACK' y 'JUMPER'.

Este apéndice contiene la dirección de las rutinas del firmware que se utilizan con mayor frecuencia. La primera columna del texto contiene las direcciones; la segunda, una breve descripción del efecto de la rutina; la tercera, los registros que modifica la rutina.

---

Dirección de llamada		Registros modificados
<b>Geslor dd teclado</b>		
BB00	Inicializa completamente el gestor del teclado	AF BC DE HL
BB12	Lee un carácter de una cadena de expansión. Entrada: A contiene el código del carácter expandible y L el número carácter que se va a leer en la cadena. Salida: A contiene el carácter leído y arrastre a 1; o bien, A corrupto y arrastre a 0 si el carácter no era expandible o la cadena no era suficientemente larga.	AF DE
BB18	Espera por una pulsación del teclado. Salida: A contiene el carácter leído y arrastre a 1.	AF
BB1B	Examina el teclado sin esperar pulsación. Salida: A contiene el carácter leído y arrastre a 1; o bien, A corrupto y arrastre a 0 si no se había pulsado ninguna tecla.	AF

---

Dirección de llamada	Función	Registros modificados
BI31E	Examina una tecla concreta. Entrada: A=n.º de tecla. Salida: indicador Z a 0 si la tecla está pulsada, y a 1 si no lo está. Siempre: arrastre a 0 y el registro C contiene el estado actual de SHIFT y CTRL.	AF HL
BB24	Examina el estado de los joysticks. Salida: A y H contienen el estado de JOY0; L contiene el estado de JOY1. Significado de los bits: 0, arriba; 1, abajo; 2, izquierda; 3, derecha; 4, disparo 2; 5, disparo 1; 6, no asignado; 7, siempre a 0.	AF HL
<b>Pantalla de texto</b>		
BB4E	Inicialización completa.	AF BC DE HL
BB5A	Envía un carácter o código de control a la pantalla. Entrada: A=código del carácter.	Ninguno
BB60	Lee en la pantalla el carácter que hay en la posición actual del cursor. Salida: si se encuentra un carácter legible, A contiene el código y arrastre se pone a 1.	AF
BB75	Coloca el cursor en la columna señalada por H y la fila señalada por L.	AF HL
La mayor parte de las restantes acciones sobre la pantalla de texto se pueden realizar escribiendo en ella códigos de control. Véase el Manual del Usuario.		
<b>Pantalla gráfica</b>		
BBBA	Inicialización completa	AF BC DE HL
BBC9	Establece el origen de coordenadas gráficas en el punto señalado por DE (x) y HL (y).	AF BC DE HL
BBDE	Asigna tinta a la pluma gráfica. Entrada: A=n.º de tinta.	AF
BBEA	Dibuja el punto de coordenadas absolutas dadas por DE (x) y HL (y).	AF BC DE HL
BBF6	Dibuja una recta desde la posición actual hasta la señalada por DE (x) y HL (y).	AF BC DE HL
BBFC	Escribe en la posición actual del cursor gráfico el carácter cuyo código está contenido en A.	AF BC DE HL
<b>Gestor de la pantalla</b>		
BBFF	Inicialización completa	AF BC DE HL
BC05	Establece la dirección de comienzo de la memoria de la pantalla. Entrada: HL=n.º de bytes en que hay que desplazar esa dirección. Este número debe ser par; la rutina lo toma MOD 80.	AF HL

Dirección de llamada	Función	Registros modificados
BC1A	Convierte las coordenadas físicas de entrada en una dirección de la memoria de la pantalla. Entrada: H = número de columna; L = número de fila. Salida: HL = dirección del extremo superior izquierdo del carácter; B = número de bytes de memoria requerido para representar un carácter en la memoria de la pantalla.	AF

Para las cuatro rutinas siguientes, el par HL debe contener la dirección de una posición de la pantalla, y el resultado se entrega en el propio par HL. Si el movimiento se va fuera de la pantalla, las rutinas no advierten de ello.

BC20	Desplaza la dirección de memoria de la pantalla un byte hacia la derecha.	AF
BC23	Desplaza la dirección de memoria de la pantalla un byte hacia a izquierda.	AF
BC26	Desplaza la dirección de memoria de la pantalla un byte hacia abajo.	AF
BC29	Desplaza la dirección de memoria de la pantalla un byte hacia arriba.	AF
BC38	Establece como colores para el borde los contenidos en B y C	AF BC DE HL
BC3E	Establece periodos de parpadeo (el contenido en H para el primer color y el de L para el segundo).	AF HL

#### Gestor del cassette

BC65	Inicialización completa	AF BC DE HL
------	-------------------------	-------------

Para manejar el magnetófono o el generador de sonidos mediante las rutinas del firmware, hay que conocerlas previamente muy a fondo. Le sugerimos que maneje estas cosas desde BASIC, aunque vuelva posteriormente al código de máquina con un CALL. Recuerde que sólo podrá volver desde código de máquina a un programa BASIC cuando provenga de dicho programa.

BD2B	Envía a la puerta Centronics (impresora) el carácter contenido en A (ignorando el bit 7). Salida: arrastre a 1 si se ha podido enviar el carácter, a 0 en caso contrario.	AF
BS37	Restaura las direcciones del grupo de saltos.	AF BC DE HL

Las rutinas que hemos presentado son sólo algunas de entre los centenares que existen. El 'Firmware Specification Manual (SOFT 158)' de Alustrad, le proporciona el detalle de todas las rutinas del firmware y una ligera explicación del hardware. Debe adquirir este manual si desea programar seriamente en código de máquina.





# índice

A 18, 20, 39-58, 80, 120, 123

a0 42

acumulador 18, 39-58, 80

ADC 39-58, 84

ADD 39-58, 61, 84

ADD A,A 96

ADD A,(HL) 44

ADD A,n 42

ADD A,r 43, 44

ADD HL,HL 103

ADD HL,SP 84

*address bus* 125-128

*add/subtract* 59-69

AF, par 80

AND 71-78

AND A 51, 56, 58

AND #DF 74

Aritmética 39-58, 84

operaciones 61

arranque en frío 29, 83, 129

arrastre 46, 71

bit de 58

indicador de 43, 58, 59-69, 78, 89,

96

ASCII 9

códigos 62

automáticas, instrucciones 113

B 18, 20, 27, 44

BASIC 141-144

BC 24, 25, 27, 80, 117, 120, 121, 123

BCD *{Binary Coded Decimal}* 111

*binary counter* 113

BIT 87-93

bit 7, 87-93

0 8, 88

7 88

más significativo 8, 28

menos significativo 28

bloques de salto 141-144

bus de datos 125-128

bus de direcciones 125-128

búsqueda, instrucción de 113-123

byte 7

C 18, 20, 27, 43, 44, 47, 59-69, 77

CALL 11, 29-33, 37, 69, 79-86

CALL 47876 65

carga 83

instrucciones de 17-29

CARGADOR HEX 11, 12, 44

*carry* 47

*flag* 43, 59-69

CCF 51, 69, 73

cero, indicador de 42, 58, 59-69, 89, 117

CLP 71-78

codificación binaria de los números

decimales 111

código(s)

ASCII 62

nemotécnicos 10

objeto 10

comparación 59-69

complementación 75

complementario, valor 51

complemento a dos 8, 67

representación en 8

- contador binario 113
- contador de programa 18, 29-33
  - posición 36
- CP 59-69
- CPD 113-123
- CPDR 113-123
- CPI 113-123
- CPIR 113-123
- CPL 75, 77
  
- D 18, 20, 27, 44
- data bus* 125-128
- DE 25, 80, 113, 116, 121, 123
- DEC 39-58, 61, 84
- DEC (HL) 39
- DEC SP 84
- decimal 7
- decisiones condicionadas 59
- DEFB 11
- DEFM 11
- DEFS 11
- DEFW 11
- desplazamiento 95-112
  - aritmético 95-112
  - lógico 95-112
- DI 129-133
- diagramas de flujo 13-15
- dirección 9
- direccionamiento indexado 135-140
- dividir 99
- división 102
- DJNZ 59-69, 113
  
- E 18, 20, 27, 44
- editor 10
- EL 129-133
- enmascarar 74
- ensamblador 10
- ENT 11
- entrada 125-128
- EQU 11
- E/S 127
- escritura transparente 75
- estado, registro de 80
  
- etiquetas 10
- EX 36, 85
- exchange 36, 85
- EX DE,HL 36
- EX (SP),HL 79-86
  
- F 42, 80
- firmware* 3
- flag* 42, 59-69
- flujo, diagrama de 13-15
- fuelle, programa 10
  
- H 18, 20, 27, 28, 37, 44, 59, 69
- half carry* 59-69
- HALT 129-133
- hexadecimal, sistema 8
- hexadecimal y binario 7
- high* 37
- HL 24, 25, 34, 37, 39, 80, 85, 113, 116, 120, 121, 123
- IM0 130
- IM1 129
- IM2 129
- IM3 129
- IN 125-128
- INC 39-58, 61, 84
- INC (HL) 39
- indicador 42, 58, 59-69
  - de arrastre 43, 58, 59-69, 78, 89, 96
  - de cero 42, 58, 59-69, 89, 117
  - de paridad 71, 78
  - de paridad/sobrepasamiento 59-69
  - P/V 71, 78, 112, 120, 123
  - de semiarrastra 59-69
  - de signo 59-69
- indicadores 71, 75, 77
- índice, registros 135-140
- indexado, direccionamiento 135-140
- input* 125-128
- intercambio 36, 85
- intérprete 3, 4
- interrupción 129-133
  - modos de 129-133
- interrupciones 129-133

instrucciones  
  aritméticas 39-58  
  automáticas 113  
  de búsqueda 113-123  
  de carga 17, 29  
  lógicas 71  
  de salto 33-36  
I/O 127  
IX 135-140  
IY 135-140

*joysticks* 143  
JP 30, 33-36, 69  
JP (HL) 37  
JP nn 37  
JR 30, 33-36, 69  
JR n 37  
*jump* 30, 33-36  
*jump-blocks* 141-144  
*jump, relative* 33-36

L 18, 20, 27, 28, 37, 44  
LD 17-29, 36, 61, 69  
LD A 17-29  
LD A,{HL} 17-29, 37  
LD A,(nn) 17-29, 37  
LD A,(rr) 27  
LD B 24  
LD BC,(nn) 28  
LD DE,(nn) 28  
LD HL 24  
LD (HL),A 17-29, 37  
LD HL,(nn) 27  
LD (HL),r 25  
LD (nn),A 17-29, 37  
LD (nn),BC 28  
LD (nn),DE 28  
LD (nn),HL 27  
LD (nn),rr 17-29, 37  
LD r,(HL) 25  
LD r,n 17-29, 36  
LD r,r' 17-29, 37  
LD (rr),A 27  
LD rr,nn 17-29, 37

LD rr,(nn) 17-29, 37  
LDD 113-123  
LDDR 113-123  
LDI 113-123  
LDIR 113-123  
listados de ensamblador 12  
llamadas 29-33  
llamar 30  
*load* 18  
lógicas, instrucciones 71  
lógicas, operaciones 71-78  
*low* 37

M 59-69  
m 112  
mandos de juego 143  
mapa de pantalla 75, 165-167  
método de restauración 109  
microprocesador Z80 2  
*minus sign* 59-69  
modo 0 75, 165-167  
modo 1 75, 165-167  
modo 2 75, 165-167  
modos de interrupción 129-133  
multiplicación 99  
multiplicar 99

N 59-69  
n 36, 58, 69, 77, 86, 112, 123  
NC 43, 59-69  
nn 36, 58, 69, 77, 86, 112, 123  
(nn) 22  
NEG 71-78  
negación 77  
negativos, números 8  
nemotécnicos, códigos 10  
NOP 50, 72  
no sobrepasamiento 59-69  
números negativos 8  
NZ 59-69

objeto, programa 10

- Operaciones
  - aritméticas 61
  - de E/S 127
  - lógicas 71-78
- OR 71-78
- OR #20 73
- OR #30 75
- ORG 11
- OUT 125-128
- output* 125-128
- overflow* 59-69
- P 59-69
- palabras 7
- pantalla 143
  - mapa de 75
- par AF 80
  - BC 25, 27, 113
  - DE 25, 27, 36
  - HL 25, 27, 36
- pares 24
- paridad impar 59-69
- paridad, indicador de 71, 78
- paridad par 59-69
- paridad/sobrepasamiento, indicador de 59-69
- parity even* 59-69
- parity odd* 59-69
- parity/overflow flag* 59-69
- Pascal 141-144
- PC 18, 20, 29-33, 35, 36, 58, 69, 77, 86, 112, 123
- PE 59-69
- pila 31-79-86
- pixels 75
- plus sign* 59-69
- PO 59-69
- POP 79-86
- port* 127
- programa
  - contador de 29-33
  - fuelle 10
  - objeto 10
- program counter* 29-33
- puerta 127
  - puntero de pila 31, 79-86
  - punto de entrada 10
  - puntos 75
  - PUSH 79-86
  - P/V 59-69, 77, 117, 118, 120, 123
    - indicador de 78
  - r 25, 36, 58, 69, 77, 86, 112, 123
  - RAM 141
  - registro (s) 133-135
    - acumulador 39-58
    - de destino 21
    - de estado 21
    - índice 135-140
    - de origen 21
  - reinicio 133-135
  - RES 87-93
  - restart* 133-135
  - restauración, método de 109
  - restoring* 109
  - RET 29-33, 37, 69, 79-86
  - retardo 133
  - RETI 130, 131
  - return* 30
  - RL 95-112
  - RLA 95-112
  - RLC 95-112
  - RLCA 95-112
  - ROM 141
  - rotación 95-112
    - circular 95-112
  - RR 95-112
  - rr 24, 27, 36, 58, 69, 77, 86, 112, 123
  - RRA 95-112
  - RRC 95-112
  - RRCA 95-112
  - RST 133-135, 142
  - RST 56 (38H) 130
  - rutina del servicio de
    - interrupciones 129-133
  - S 59-69, 77
  - salida 125-128
  - salto 30, 33-36
    - instrucciones de 33-36

Salto	SUB n 42
magnitud 37	SUB r 43, 44
relativo 33-36, 37	suma/resta, indicador de 59-69
SBC 39-58, 61, 84	
SCF 51, 69	
SET 87-93	transferencia de bloques 113-123
seudo-operaciones 11	TXT OUTPUT 30
<i>signflag</i> 59-69	
signo	
indicador de 59-69	valor complementario 51
positivo 59-69	volver 30
negativo 59-69	vuelco de pantalla 138
sin signo 8	
sistema	
hexadecimal 8	WAIT KEY 65
operativo 3, 141, 144	
SLA 95-112	
sobreescritura 74	XOR 71-78
sobrepasamiento 59-69, 71, 77	XOR #FF 75
indicador de 59-69	
sonido 143	
SP 31, 36, 58, 69, 77, 79-86, 112, 23	Z 59-69, 77
SRA 95-112	<i>zero flag</i> 42, 59-69
SRL 95-112	Z80, microprocesador 2
<i>stack</i> 79-86	
<i>stack pointer</i> 31, 79-86	
SUB 39-58	( ) 36, 58, 69, 77, 86, 112, 123
SUB (HL) 44	\$ 36

El Amstrad CPC464 es probablemente la novedad más importante en ordenadores desde la aparición del Spectrum. Su BASIC incorpora muchas funciones avanzadas que antes sólo se podían encontrar en ordenadores de precio mucho mayor.

Este libro va dirigido al principiante que desea aprender a programar en código de máquina en el Amstrad CPC464. Empieza por presentar los conceptos de programación en código de máquina, explica las instrucciones que el Z80 entiende y cómo utilizarlas, y describe algunas rutinas del sistema operativo. En el libro se incluyen algunos programas que facilitan la introducción de programas en código de máquina, así como la inspección, la modificación y el desplazamiento del contenido de zonas de memoria. Las rutinas del sistema operativo han sido utilizadas con frecuencia, lo que permite ver inmediatamente los resultados de los programas.

### El autor

Steve Kramer ha estado trabajando en informática durante estos últimos doce años. En la actualidad es consultor de varias casas de programación en el campo de las aplicaciones comerciales. También es autor de diversos juegos y programas para los microordenadores más populares.

**AMSTRAD**  
**E S P A Ñ A**

Avd. del Mediterráneo, 9 28007 MADRID

